



Calhoun: The NPS Institutional Archive
DSpace Repository

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

2000

Requirements reuse in support of the aviation mission planning system migration to the Joint Mission Planning System.

Stierna, Eric J.

Monterey, California. Naval Postgraduate School

<http://hdl.handle.net/10945/32963>

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

<http://www.nps.edu/library>

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

**REQUIREMENTS REUSE IN SUPPORT OF THE
AVIATION MISSION PLANNING SYSTEM MIGRATION
TO THE JOINT MISSION PLANNING SYSTEM**

By

Eric J. Stierna

September 2000

Thesis Advisor:
Thesis Co-Advisor:

Man-Tak Shing
Neil Rowe

Approved for public release; distribution is unlimited.

20001130 052

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 2000		3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE Requirements reuse in support of the Aviation Mission Planning System migration to the Joint Mission Planning System			5. FUNDING NUMBERS	
6. AUTHOR(S) Stierna, Eric J.				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) Developing correct, complete, consistent and clearly defined requirements is expensive and time-consuming, but is critical to the success of software development. Existing written requirements represent a vast source of domain knowledge that a software analyst can extract for the design of new systems. This thesis describes a modeling process and tool set to identify similar requirements in two requirement documents. We tested our methods in a comparison of the Aviation Mission Planning System (AMPS) legacy software and the new Joint Mission Planning System (JMPS). Our analysis process creates domain entities, a requirements repository, and statistical matching information for a domain analyst to evaluate reuse potential. We automated several key tools. Our results showed that the proposed process and tools significantly shorten the time needed to reuse software requirements.				
14. SUBJECT TERMS Requirements Reuse, Keyword Matching, Aviation Mission Planning System, Joint Mission Planning System, Domain Modeling, Domain Analysis			15. NUMBER OF PAGES 128	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

Approved for public release; distribution is unlimited

**REQUIREMENTS REUSE IN SUPPORT OF THE AVIATION MISSION PLANNING
SYSTEM MIGRATION TO THE JOINT MISSION PLANNING SYSTEM**

Eric J. Stierna
Captain, United States Army
B.A., Brown University, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

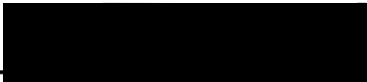
from the

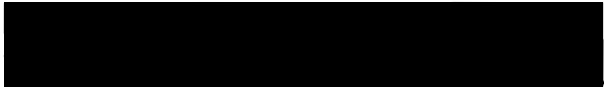
**NAVAL POSTGRADUATE SCHOOL
September 2000**

Author


Eric J. Stierna

Approved by:


Man-Tak Shing, Thesis Advisor


Neil Rowe, Thesis Co-Advisor



Dan Boger, Chairman
Department of Computer Science

TABLE OF CONTENTS

I. INTRODUCTION.....	1
A. REQUIREMENTS REUSE PROBLEM	3
B. THESIS OBJECTIVES AND SCOPE	5
C. APPROACH	5
D. STRUCTURE.....	6
II. BACKGROUND	7
A. DOMAIN ENGINEERING.....	7
B. REQUIREMENTS ENGINEERING	10
C. REQUIREMENTS REUSE.....	11
D. THE REUSE ENVIRONMENT.....	12
E. DOMAIN THEORY.....	13
F. THE REUSE TOOLSET.....	15
1. <i>Application Generators</i>	15
2. <i>Reuse Libraries</i>	15
3. <i>Domain Modeling Tools</i>	16
4. <i>Software Engineering Environments (SEE)</i>	16
5. <i>Formal Methods and Informal Descriptions</i>	17
G. XML.....	17
H. SPEC.....	18
1. <i>Original Requirement</i>	19
2. <i>Parsed Requirement</i>	19
3. <i>Selected Entity</i>	20
4. <i>Definition/Sense Query</i>	20
5. <i>Select Sense/Definition</i>	20
6. <i>Spec Module</i>	20
7. <i>SpecXML Document Format</i>	21
III. RESEARCH APPROACH.....	25
A. PROBLEM	25
IV. MANUAL MATCHING METHODOLOGY	27
A. SELECT THE DOMAIN	28
B. COLLECT THE REQUIREMENTS	29
C. REFINE THE DOMAIN DEFINITION	30
D. IDENTIFY THE MATCHING REQUIREMENTS.....	30
1. <i>Evaluate Each Requirements Document for Composite Requirements</i>	31
2. <i>Partition the Composite Requirements</i>	31
3. <i>Select Criteria for a Match</i>	32
4. <i>Compare the Requirements Documents</i>	33
5. <i>Identify Matches</i>	33
E. IDENTIFY OVERLAPPING REQUIREMENTS	35
F. IDENTIFY UNMATCHED REQUIREMENTS.....	35
G. VALIDATE THE REQUIREMENT MATCHES AND OVERLAPS WITH THE STAKEHOLDERS.....	36
H. INTEGRATION OF REQUIREMENTS MATCHES AND OVERLAPS INTO A DOMAIN MODEL	36
V. EVALUATION OF THE MANUAL MATCHING PROCESS.....	39

VI. AUTOMATED MATCHING METHODOLOGY.....	43
A. DEVELOPMENT GOALS.....	43
B. LANGUAGE	44
C. PARSING PREPARATION.....	44
D. WORD MATCHING	45
E. REFINEMENTS TO BASIC WORD MATCHING.....	47
F. MACROREQUIREMENTS.....	48
VII. EVALUATION OF THE AUTOMATED MATCHING PROCESS.....	49
A. WORD MATCHING	50
B. COMBINED FILTER MATCHING	51
C. MACROREQUIREMENT BUNDLED MATCHING	52
D. RESULTS.....	53
VIII. CONCLUSIONS & FUTURE WORK.....	55
APPENDIX A: MANUAL MATCHING PROTOTYPE.....	59
APPENDIX B: MANUAL MATCHING RESULTS.....	63
APPENDIX C: AUTOMATED TOOL SOURCE CODE.....	65
LIST OF REFERENCES.....	113
INITIAL DISTRIBUTION LIST	117

I. INTRODUCTION

Whether formally or informally specified, requirements describe what is to be built in software or hardware. Correct, complete, consistent and clearly defined requirements are critical to successful system development.

Requirements exist as a result of system stakeholders conducting analysis to identify their needs for a system to perform a given task. Requirements exist in the context of the problem domain in which they are formulated. This problem domain contains the external objects with which a system must interact, concepts and knowledge that apply to the problem, and the stakeholders that maintain perspectives on all domain entities [BJOR98].

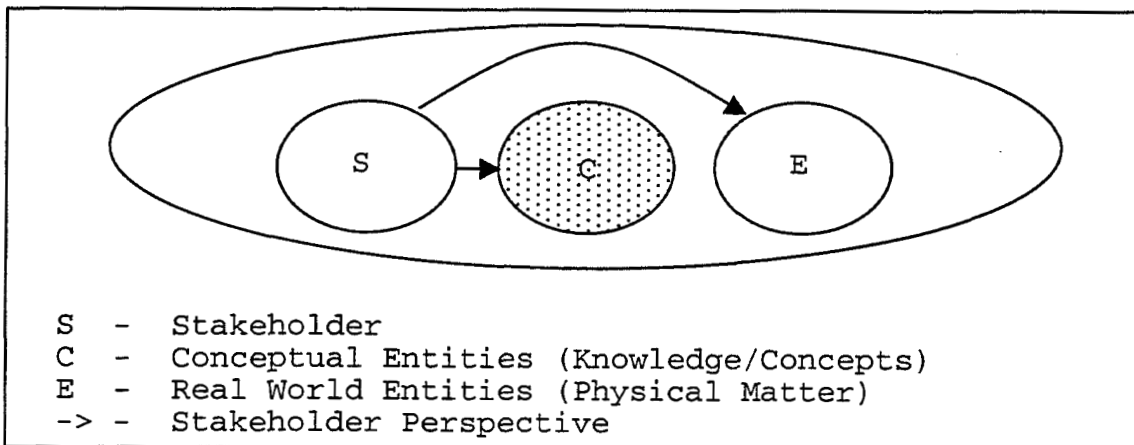


FIGURE 1-1. Problem Domain

The scope of a problem domain depends on the domain complexity, stakeholder understanding, and physical medium.

Building cost-effective systems using software to solve problems with a large scope involves a multi-faceted array of management and technical expertise. Solutions to the problem of consistently providing such systems have been implemented with varying degrees of success [DAVI94]. A constant thread in each solution is the importance of getting the requirements correct.

Domain engineering and requirements engineering address this issue through a variety of approaches to elicit, define, and analyze requirements in the context of the problem domain [KOTO98]. These approaches include a variety of engineering processes that use modeling, structured analysis, object-oriented analysis, and other methods to create correct, well-documented models and requirements specifications. Requirements represent a large source of domain knowledge that can be exploited for reuse, to improve the quality of the requirements of future software systems [BJOR98].

In the face of dramatic improvements in computer hardware, shortages in developer manpower, and increasing demand for software, an ever-widening gap between inexpensive hardware and costly software development has formed. This gap creates a real need for improved processes and tools to produce cost-effective software. Adaptable requirements-reuse processes appear to be an effective

technique to provide software on the scale and at the speed needed to meet large-scale software demands. Bjorner [BJOR98] calls for domain models that allow non-proprietary sharing of information about problem domains in order to lay a foundation for a long-term solution.

The Department of Defense (DOD) uses and develops large-scale, complex, software-based systems to accomplish mission-critical tasks. Various agencies within DOD have initiated domain-analysis and requirements-analysis efforts such as STARS [SOLD92] and FODA [COHE92], but smaller organizations have failed to integrate these experiences into their software-development efforts.

A. REQUIREMENTS REUSE PROBLEM

Requirements reuse is important to the United States Army Electronics Command (AEC) as it relates to the mission to migrate the legacy Aviation Mission Planning System (AMPS) software to interoperate with the future Joint Mission Planning System (JMPS). The JMPS system is a distributed mission-planning application developed using a product-line approach in accordance with the latest DOD interoperability requirements. The legacy AMPS software system has requirements for the same mission-planning domain as the JMPS system. However, AMPS is a system with a closed architecture that contains Army-specific and aircraft-specific requirements not in the set of JMPS requirements.

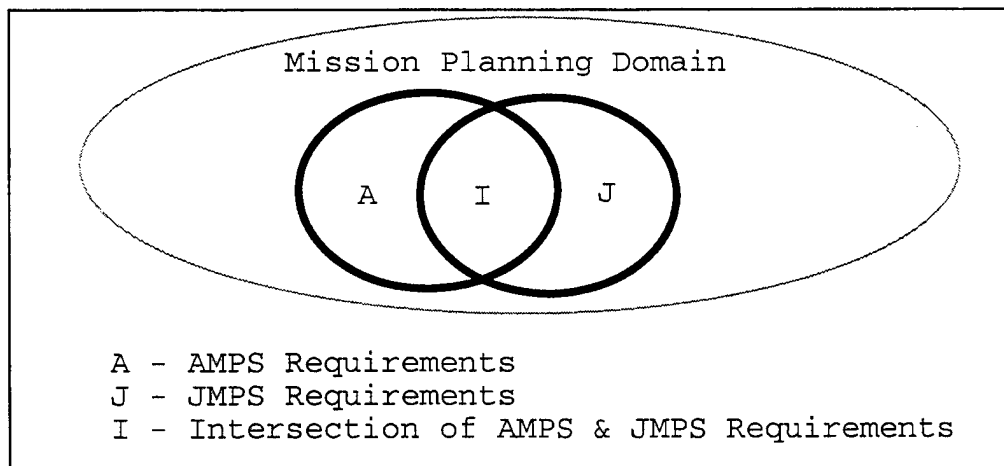


FIGURE 1-2. Mission Planning Domain

Given the 1998 congressional mandate to combine the 41 different DOD mission-planning systems into the JMPS architecture [WALE99], AEC must migrate its software to the new mission-planning architecture. Migration of legacy mission-planning systems to a new software architecture in a Joint (Multi-Service) environment introduces a set of risks to the software-development process that must be evaluated. Evaluation of migration difficulty involves determining the similarities and differences between requirements. Current procedures for matching requirements between two systems are labor-intensive and rely on extensive domain expertise of the user and developer. A simpler, repeatable process that improves an analyst's ability to reuse requirements from legacy systems is desirable. In addition, the ability to

match requirements to legacy systems reduces the errors in developing new systems.

B. THESIS OBJECTIVES AND SCOPE

This thesis develops both a manual process and tool to automate the identification of common requirements in two requirement documents. The outputs of both are reports that detail the requirements overlap between the two systems. We also identified a persistent set of domain entities ready for integration into a domain model. The goal of this thesis work is to provide assistance for requirements reuse in the Joint military mission-planning domain using the best practices from software engineering.

C. APPROACH

We used a manual matching process based on guidance received combat developers at AEC [WALE99] to establish initial pairs of matched requirements. We then used the insights gained in that process to develop a tool to partly automate requirements reuse. The Java-based tool extracts requirements systematically for an analyst with experience in the domain. The tool matches words between pairs of requirements and calculates a similarity rating based on word statistics. The tool provides the option to transform extracted requirements and domain entities into XML text files for integration into a reusable domain model.

D. STRUCTURE

The thesis is organized into eight chapters:

Chapter II provides background information about domain engineering, domain analysis, requirements engineering, requirements reuse, and XML. This information supports the problem analysis and research approach.

Chapter III describes the research methodology used in the thesis.

Chapter IV presents the manual matching process methodology.

Chapter V discusses the results of the manual matching process on the two requirements documents.

Chapter VI presents the automated matching tool (AMT) development methodology.

Chapter VII discusses the results of the automated matching tool on the two requirements documents.

Chapter VIII presents conclusions and future Work.

Appendix A presents the summary of the manual matching prototype study. Appendix B presents a spreadsheet summary of the manual matching study. Appendix C contains the source code for the automated matching tool.

II. BACKGROUND

This chapter provides overview information on domain engineering, requirements engineering, requirements reuse, formal and informal analysis methods, formal specification languages, modeling languages, and mark-up languages as they pertain to this thesis.

Incorrect system requirements have plagued the development of software systems since the 1960s. Failing to provide desired functionality within budget and on time has been all too common for most systems developed in the last 40 years [KOT098]. This has played an important role in advancing the current practice of domain engineering and requirements engineering.

A. DOMAIN ENGINEERING

Domain engineering develops a precise description of a real-world context in which a particular problem must be solved. The stakeholders and the domain analysts negotiate on a well-defined set of descriptions that carefully describe the environment in which the stakeholders' requirements exist [BORJ98]. These descriptions can be modeled formally or informally for the analysis phase of a spiral development process.

Examples of problem domains where domain engineering has been important in development of software systems are:

- Transportation Systems (rail, air, bus, maritime);
- Manufacturing (marketing, production, storage);
- Financial Services (banking, insurance, securities);

and

- Health Care (medications, procedures, long-term care)

[BJOR98].

Domain engineering looks at problem instances and uses reasoning skills to create a generic architecture and model that describes the problem environment (domain) in an abstract fashion. This includes domain analysis, domain architectures, and domain abstractions.

The link between a system's requirements and its domains is important. Many requirements specify system constraints and operations derived from the domain rather than system functionality [KOT098]. For example, requirements for a particular flight simulator specify the mass of the aircraft, its thrust-to-weight ratio, minimum runway length, and minimum airspeed needed for takeoff. Specification of this information is needed for all flight simulators. This type of domain information can be derived from a legacy system's requirements in many instances.

"Domain" has many meanings. For domain analysis, Jackson [JACK95] defines it as "a general class of systems for an application area such as resource management, or airline reservations, or banking, or production control."

The application domain contains the entities (objects), concepts, and constraints that apply to it. Esprit Inc. [FRAN97] divides domains into three categories: real-world domains, technology domains, and axiomatic domains. Real-world domains concern policies, behaviors, and sociological conventions that people use to interact. Technology domains concern automation of a real-world manual activity or natural process. Axiomatic domains focus on the key patterns of system behavior that are common to a class of applications. An example of an axiomatic domain is boundary-condition monitoring; partitioning it from the technological domain of a particular system with specific boundary thresholds. This creates a reusable pattern that can influence a class of systems.

The purpose of modeling a domain is to facilitate knowledge reuse. Software evolution involves making reliable changes to legacy software to add new functionality and quality of service. This change involves risks that include unexpected or undetected consequences, degradation of design quality, and increased costs [ARAN93]. The lack of reusable knowledge is estimated by Arango [ARAN93] to account for 35% to 80% of evolution costs and for most of the risk. Luqi [LUQI97] cites 180 billion US dollars spent in 1996 on software development projects that were curtailed or terminated due to lack of domain understanding or incorrect

requirements. In fact, the larger the scope of a problem and the larger the programming team, the greater the need for domain models to standardize vocabulary, ensure that the problem is well understood, and ensure that requirements are well defined in the domains.

B. REQUIREMENTS ENGINEERING

Requirements engineering involves discovering, documenting, and maintaining a set of requirements for a computer-based system [KOTO98]. Its goal is precise partial specifications of the domain in the form of a type space and a set of functions for the functional requirements as well as non-functional requirements like quality of service, machines, and domain/machine interfaces [BJOR98].

Requirements engineering can also be defined as the systematic and repeatable techniques used to discover, document, resolve ambiguity and inconsistency between conflicting views, and maintain requirements for a computer-based system [KOTO98]. These techniques may involve formal requirements engineering that consists of the derivation, validation, creation, and maintenance of a requirements document for a given domain or application instance. Outputs of requirements engineering include process models, requirements documents, software-specification documents, requirements-management techniques, and requirements-modeling techniques [KOTO98].

C. REQUIREMENTS REUSE

This thesis will examine a sub-problem of requirements engineering known as requirements reuse. This is the process of reusing requirements from previous systems or domains to develop a new system. Sommerville and Sawyer [SOMM97] distinguish direct and indirect reuse. Direct reuse is insertion with minimal change into the requirement set of a new system. This is difficult due to subtle domain differences that may not manifest themselves immediately. Proprietary knowledge issues may prevent direct reusability of a requirement. For example, a proprietary quality-of-service timing constraint requirement could not be reused in a non-proprietary requirement document. Indirect reuse is less difficult and uses an existing requirement to elicit, analyze, and validate a requirement for a new system. It involves a dialog with a user or a domain analyst and can take the form of an automated matching process to identify common domain terms, a guided discussion with users and analysts to develop a set of requirements and goals in Object-Oriented Analysis/Design, or an unstructured elicitation process with users providing requirements.

Requirements reuse offers the potential to save money and time by capitalizing on an existing knowledge. Kotonya and Sommerville [KOTO98] claim that 50% of all requirements may be the same for similar systems in similar domains.

Requirements reuse is normally part of the requirements elicitation phase of the development process. It may be part of domain analysis, requirements definition, cost estimation, or feasibility analysis [SOMM97]. It can impact all phases of the development. Sommerville and Sawyer [SOMM97] recommend a small team of two or three people to develop a reuse program within a software development organization.

D. THE REUSE ENVIRONMENT

Software reuse involves:

- Employing existing assets in the software-product development process, while preserving asset integrity; and
- Application of existing solutions to problems of systems development [LIM98].

Commercial practice has addressed the problem with the product-line approach [BAT098], and domain-specific proprietary software environments [BJOR98]. Academic research addressed the problem with strategies and tools that are covered in the remainder of this chapter. However, no single process or approach applies to all reuse situations.

Software reuse has three parts: acquisition of a reusable component, representing the component in a given form, and reuse of the component to solve a particular

problem [MAID91b]. Requirements reuse is a subset of software reuse, but requirements are valuable both for the information they contain and their linkage to the other components in the development process. A requirement document presents a lucrative environment for the extraction of the domain objects, functions, data, and states [DAVI94]. Textual descriptions of the problem domain are used in the conceptual modeling process [LARM97]. Direct noun-to-concept mapping is rarely possible due to natural-language ambiguity. However, natural-language requirements documents still can yield valuable information for reuse. Arango, one of the pioneers in software reuse, proposed that a reuse infrastructure be available to the developer, obtained through incremental domain analysis [ARAN89].

E. DOMAIN THEORY

Domain theory defines the semantic context, boundaries, and granularity of a given software-engineering abstraction [SUTC98][MAID94a]. It uses models of human reasoning and memory, a class-hierarchy structure, and the concept of generic classes to describe the problem environment. Key concepts of domain theory are knowledge metaschema, domain abstractions, and matching processes.

A knowledge metaschema is a modeling language that defines the semantics of generic classes. Examples of its semantic primitives are key objects, agents, structure

objects, state transitions with respect to structure objects, states, goals, activities, object properties, events, state conditions (pre/post) and relationships [SUTC94].

Domain abstractions represent the fundamental behavior, structure, and functions of a class of domains [MAID94]. Abstractions divide the domain-analysis task into subsections to simplify the automated analysis of requirements. Sutcliffe and Maiden divide domain abstractions into two model types. Object-systems models represent domain structure and behavior; objects have properties and states that can be affected by physical, financial, and conceptual laws [SUTC94b]. Information-systems models specify processes for report production, summaries, progress checking, object queries, count, and list functions [MAID94a].

A software-component matching process is also a key element in a requirements-reuse effort. The process of finding the right match can be carried out formally or informally or with a combination of techniques. The list of methods found in the literature include navigation (browsing), keyword search, query, dialog-assisted search, dialog-specified search, analogical matching, and case-based reasoning [DAVI94][MAID93a][MAID91a].

Requirement matching identifies shared requirements between two or more systems. These matched requirements contain entities that define the core domain objects and concepts. Analysts can use the entities to develop domain models, identify potential product-line applications, and determine variation between requirements.

F. THE REUSE TOOLSET

Many tools are available for domain analysis and requirements engineering.

1. Application Generators

Application generators use design decisions of an applications engineer for a well-specified domain to retrieve relevant components in a software repository [DAVI94]. An example of this is the CAPS prototype which takes a problem specified in the PSDL language and uses a repository of Ada modules to create an executable skeleton for the application [IBRA96]. Application generators typically affect reuse in the design-reuse phase rather than the requirements-definition phase.

2. Reuse Libraries

Reuse libraries are collections of software resources and related documentation designed to aid in software development, reuse, and maintenance [LIM98]. Some libraries act as domain database repositories like those used by Sutcliffe [SUTC94a], Maiden [Maid94a], and Bjorner [BJOR98].

3. Domain Modeling Tools

Domain modeling tools help partition the problem domain. An example is the DARE-COTS (Domain Analysis and Reuse Environment) CASE tool used in the STARS program [FRAK97]. It provides mechanisms for extracting and recording domain knowledge from documents, code, and human experts. It performs analysis on the acquired knowledge to generate domain models, and creates repositories of reusable assets for the given domains [FRAK97]. Related tools are the Software Engineering Institute's Feature Oriented Domain Analysis (FODA), Organon Motives' Organization Domain Modeling (ODM), and the Paramax Systems Corporation's READS tool [SMIT92].

4. Software Engineering Environments (SEE)

Another approach is a software-engineering environment (SEE) to support a product-line approach to development [HAMI93]. Its key elements are:

- Automation and tooling to support process definition and modeling;
- Automation and tooling to support domain-specific reuse;
- Flexible framework services to support tool integration and interoperability; and
- Standards for tool interoperability across hardware platforms.

5. Formal Methods and Informal Descriptions

Problem-domain models, goals, requirements, and functional specifications may contain a wide range of formal or informal descriptions. Stakeholders use natural-language descriptions, mathematical equations, and diagrammatic descriptions for this. These descriptions contain the key objects, concepts, and relationships that define a problem [BERZ91]. Well-written formal descriptions provide precise, unambiguous definitions of the laws, objects and relationships. This structure gives an analyst the ability to develop tools to perform automated model checking [GREE94] and develop requirements based on a well-defined problem space.

G. XML

Extensible Markup Language (XML) is a subset of Standard Generalized Mark-up Language (SGML). It can describe a class of data objects and partially describe the behavior of applications that use those objects described in ISO 8879:1986(E) [W3C98]. XML documents contain parsed and unparsed data. The parsed data can be divided into character data and markup instructions. XML documents can be scanned to verify that a document conforms to a given specification and that the document is well-formed syntactically [EDDY99]. XML is under development by the XML

Working Group, which is a part of the World Wide Web Consortium (W3C).

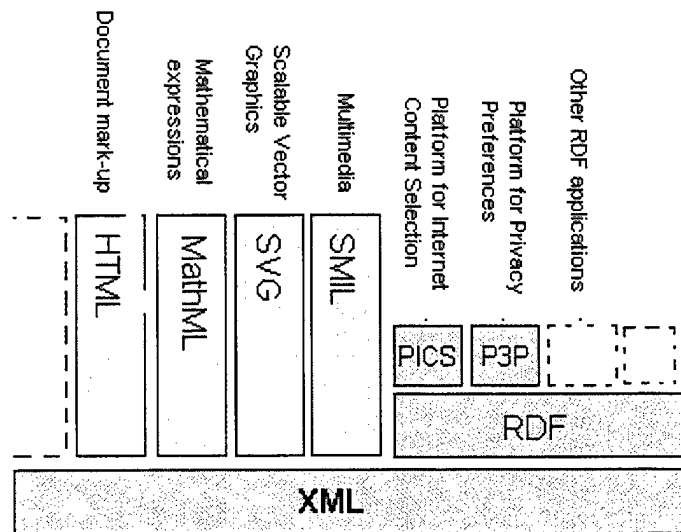


FIGURE 2-1. XML Structure From
[<http://www.w3.org/XML/Activity.html>]

H. SPEC

Spec is a language for formal modeling of domain objects, functions, types, and machines. The Spec Bachus Naur Form (BNF) when used in conjunction with an XML parser and Document Type Declaration (DTD) provides a well-formed and validated structure that ensures the correct syntax for domain entities. This facilitates the use of advanced tools to analyze domain models using theorem proving and model checking [CLAR96].

Transforming a natural language requirement into a domain entity expressed in XML involves a number of steps:

- Capturing the original text;
- Parsing it into key elements;
- Selecting an entity;
- Defining the entity;
- Selecting a domain sense;
- Building a Spec module; and
- Converting to an XML document.

Requirements can be captured with a parser designed to recognize their syntax. The analyst then selects an entity, defines it manually or with a definition database query, assigns a sense, and begins construction of the Spec module. An example from the JMPS SSS document demonstrates the steps.

1. Original Requirement

"JMPS-018-02000. JMPS shall display the locations of enemy airfields on any GI&S background or layers using MIL-STD-2525B symbology."

2. Parsed Requirement

Requirement Number: JMPS-018-02000

Requirement Text: "JMPS shall display the locations of enemy airfields on any GI&S background or layers using MIL-STD-2525B symbology."

Key Terms: display locate enemy airfield GI&S
background layer MIL-STD-2525B symbology

3. Selected Entity

Domain Entity: airfield

4. Definition/Sense Query

Definition: (Query result from WordNet1.6 Database)

The noun airfield has 1 sense (no senses from tagged texts)

1. airfield, landing field, flying field, field -- (a place where planes take off and land)

5. Select Sense/Definition

SELECTED SENSE: Sense 1

PART OF SPEECH: Noun

DEFINITION: a place where planes take off and land

SYNONYMS: landing field, flying field, field

DEFINITION/SENSE DATABASE(S): WordNet 1.6

6. Spec Module

DEFINITION airfield

CONCEPT airfield: type

CONCEPT name: type

END

7. SpecXML Document Format

```
<?xml version="1.0" encoding="UTF-8"?>
<DOCTYPE definition_declaration SYSTEM "SPEC DTD.dtd">

<definition_declaration>

    <module_DEFINITION_keyword>
        DEFINITION
    </module_DEFINITION_keyword>

    <interface>
        <NAME> airfield </NAME>
    </interface>

    <module_concept>
        <module_CONCEPT_definition_keyword>
            CONCEPT
        </module_CONCEPT_definition_keyword>
        <module_CONCEPT_NAME>
            airfield
        </module_CONCEPT_NAME>
        <assignment_operator> : </assignment_operator>
        <type_spec>
            <parameterized_name>
                <NAME> type </NAME>
            </parameterized_name>
        </type_spec>
    </module_concept>

    <module_concept>
        <module_concept_definition_keyword>
            CONCEPT
        </module_concept_definition_keyword>
        <module_concept_name> name </module_concept_name>
        <assignment_operator> : </assignment_operator>
        <type_spec>
            <parameterized_name>
                <NAME> type </NAME>
            </parameterized_name>
        </type_spec>
    </module_concept>

    <END_definition_keyword> END </END_definition_keyword>

</definition_declaration>
```

FIGURE 2-2. SpecXML Document

Spec may be used to describe the definitions and concepts that make up a domain. A Spec module is derived from a problem statement. The problem statement can be broken into a set of goals that are transformed into an environmental model through the use of DEFINITION and CONCEPT keywords. Elaboration of the concepts can be contained in the definition module or external modules. An example of a problem statement from the C4ISR domain highlights the method:

Determine the factors impacting software-based aircraft tracking systems used in the United States military. The goal is to identify the set of services provided by the legacy systems in order to develop a new system that can be configured to support the needs of each service

FIGURE 2-3. C4ISR Problem Statement Extract

An extract of the high-level goals could be expressed in Spec as:

```
-- G1 The tracking system should detect targets using sensors
DEFINITION goals

    INHERIT sensor_concept_environment
    INHERIT tds_environment
    INHERIT tactical_context_environment

    CONCEPT receives_nav_reports: boolean
        WHERE receives_nav_reports <=> SOME(ns: navigation_sensor :
gives_to(ns, nav_report, tds)),
        Subtype(receives_nav_reports, activity),
        periodic(receives_nav_reports),
        period(receives_nav_reports) <= 5 * second,
        goal(receives_nav_reports, tds)

-- G1.1 The tracking system should update its display

    CONCEPT display_tds_position: boolean
        WHERE display_tds_position <=>
        Subtype(display_tds_position, activity),
        periodic(display_tds_position),
        period(display_tds_position) <= 5 * second,
        displayed_to(location(nav_report), user),
```

```
        displayed_to(heading(nav_report), user),  
        displayed_to(speed(nav_report), user),  
        goal(display_tds_position, tds)  
  
-- G1.2 The tracking system should update location  
    CONCEPT update_vicinity: boolean  
        WHERE update_vicinity <=> updates(operator, tds,  
        vicinity), goal(update_vicinity, tds)
```

FIGURE 2-4. Spec Goals

THIS PAGE INTENTIONALLY LEFT BLANK

III. RESEARCH APPROACH

We used a spiral development process to answer the questions posed by this thesis. Our research identified requirements, developed prototype processes and tools, tested the prototypes to validate their output, and repeated the process. The next chapters cover the steps.

A. PROBLEM

This thesis addresses four questions:

- What kind of repeatable process can determine the matching requirements, the partially overlapping requirements, and the unmatched requirements that exist between two requirements documents?

- Can an analyst's tool demonstrate greater than 20% reduction in time needed to determine matching requirements over a manual process?

- Can an extendible technique or tool use matching data to provide useful input to a domain model?

- Specifically, how well do the JMPS system requirements satisfy the AMPS system requirements?

THIS PAGE INTENTIONALLY LEFT BLANK

IV. MANUAL MATCHING METHODOLOGY

Our approach had five stages: Analyze the problem; develop a standardized manual-matching process; develop an automated tool to simplify manual matching; evaluate the performance improvements; and implement a method to incorporate results into a domain model.

Initial problem analysis included meetings with sponsors from PM AEC, conversations with domain experts within the mission planning field, and development of a milestone chart with interim objectives. We adapted the high-level milestones from a collection of sources [ARAN91] [FRAN95] [SOMM97]:

- Select the domain;
- Collect the requirements;
- Refine the domain definition;
- Identify the matching requirements between the two requirement sets;
- Identify overlapping (partially matching) requirements between the two requirement sets;
- Identify requirements that are unique to one of the two systems;
- Validate the requirements matches and overlaps with the stakeholders; and
- Integrate the requirements matches and overlaps into a domain model.

A. SELECT THE DOMAIN

We studied the domain of military-mission planning and the sub-domain of automated mission-planning. Specifically, we considered Army Aviation mission-planning and Joint mission planning. Each has elements within the real-world, technological, and axiomatic domains. The domain boundaries, scope, and level of granularity were selected based on meetings with the senior project engineer [WALE99].

Analyzing the domains required understanding each mission-planning environment in sufficient detail to evaluate the context of requirements. Jackson [JACK95] describes the first step in this process as structuring and analyzing the application domain. Our research was expedited by a large body of knowledge in military doctrinal publications such as MIL-STD-2525A [MILS96] and key acquisition documents [JMPS99] [AMPS97] [AORD97].

The Aviation Mission Planning System (AMPS) is a software-based mission-planning tool that automates aviator mission-planning tasks. It can improve battlefield synchronization, intelligence, and command-and-control through communication with the Aviation Tactical Operations Center (AVTOC) and the Army Airborne Command and Control System (A2C2S). At the crew level, AMPS generates mission information for pilots in hard-copy and electronic formats for upload to aircraft via a Data Transfer System Cartridge.

AMPS has been proven to reduce the error, time and workload currently associated with pre-mission planning and aircraft subsystems' initialization tasks [AMPS97].

In contrast, the Joint Mission Planning System (JMPS) is a more general system that provides scaleable mission-planning software that can be tailored for specific needs, supports a range of hardware, provides collaborative inter-service mission planning, and enables information exchange for geographically distributed users. The JMPS architectural framework supports the development and maintenance of mission-planning components for new, modified, and improved weapon systems and operational protocols. The JMPS system is a superset of the Air Force, Navy, and Marine Corps mission planning systems [JMPS99].

B. COLLECT THE REQUIREMENTS

The requirements for the AMPS system comprise an Operational Requirements Document (ORD) [AORD97], and a System Sub-System Specification (SSS) [AMPS97]. The JMPS requirements comprise an SSS [JMPS99], an external Interface Requirements Specification (IRS) [JMPS99a], a Concept of Operations (CONOPS) [JMPS99a], Use Cases [JMPS99a], and Scenarios [JMPS99a]. All requirements were available in electronic format. No System Requirements Specification (SRS) was available for either system at the time of the study. The primary sources of requirements documents were

the AMPS project office [WALE99] and the JMPS program web site [JMPS99a].

Initial requirement collection was conducted over four weeks. We met stakeholders from the AMPS, JMPS, and developer teams to determine their views of the problem domain, obtain the required domain information, and develop a collection strategy to ensure that we gathered a sufficient set of requirements. This phase was important in identifying the stakeholders responsible for writing the requirements documents.

C. REFINE THE DOMAIN DEFINITION

Based on the research objective to determine matching requirements, we decided to restrict the domain to entities in the requirements documents. This does not support a matching set of mission-planning requirements across all systems, but it is a large step toward that goal.

Since the SSS for each system was written at a similar level of abstraction, we matched at the SSS level. Selecting comparable requirements was key to meaningful matching.

D. IDENTIFY THE MATCHING REQUIREMENTS

These steps were used to identify matching requirements:

- Search each requirements document for composite requirements;
- Partition the composite requirements along sub-domain concepts;
- Select criteria for a match;
- Compare the requirements documents by selecting a base document and a match document, and then iterating through each requirement in the base document, evaluating its degree of similarity to each requirement in the match document; and
- Record matches with a high degree of similarity.

1. Evaluate Each Requirements Document for Composite Requirements

The AMPS SSS document (44 pages/577 requirements) contained fewer requirements than the JMPS SSS document (303 pages/3538 requirements), but more of the AMPS requirements were composites.

2. Partition the Composite Requirements

We evaluated the document in the context of the real-world, technological, and axiomatic domains, reviewed the partitioning of the document, and identified the methods used to organize the requirements. This involved reading similar requirements to gain a sense of the high-level concepts and highlighting key words unique to particular domains. We divided requirements along technological and axiomatic differences, as shown in Figure 4-1.

Original Requirement

3.1.2.3.1 Protocols. To support Standardization, Interoperability and Commonality, AMPS must be capable of sharing data with other users, platforms and military services. Therefore, the AMPS shall provide the capability to format, read, interpret and display data/files via the requisite formats of the following protocols:

- * Ethernet, IEEE 802.3
- * MIL-STD-1553
- * ATHS/IDM/TACFIRE
- * MIL-STD-188-220 {Variable Message Format (VMF)}
- * MTS
- * File Transfer Protocol (FTP)
- * Remote Copy (RCP)
- * Distributed Computing Environment

Decomposed Requirements

3.1.2.3.1 "Protocols. To support Standardization, Interoperability and Commonality, AMPS must be capable of sharing data with other users, platforms and military services. Therefore, the AMPS shall provide the capability to format, read, interpret and display data/files via the requisite formats of the following protocols:"

- | | |
|--------------|---|
| 3.1.2.3.1.1 | Ethernet, IEEE 802.3 |
| 3.1.2.3.1.2 | MIL-STD-1553 |
| 3.1.2.3.1.3 | ATHS |
| 3.1.2.3.1.4 | IDM |
| 3.1.2.3.1.5 | TACFIRE |
| 3.1.2.3.1.6 | MIL-STD-188-220 {Variable Message Format (VMF)} |
| 3.1.2.3.1.7 | MTS |
| 3.1.2.3.1.8 | File Transfer Protocol (FTP) |
| 3.1.2.3.1.9 | Remote Copy (RCP) |
| 3.1.2.3.1.10 | Distributed Computing Environment |

FIGURE 4-1. Composite Requirement Decomposition

3. Select Criteria for a Match

Identifying a match pair involves determining if there is sufficient semantic similarity to ensure that one

requirement's meaning is fully captured in the other requirement. The determination involves:

- reading each requirement in the context of a document's problem domain;
- evaluating the document's structure and format;
- partitioning the document into logical sub-domains;
- evaluating the content of requirements that precede and follow an evaluated requirement; and
- interpreting the evaluated requirement's actual content.

Three to five keywords from each requirement were chosen to represent the key domain concepts or entities. These domain keywords were used to help find other related requirements.

4. Compare the Requirements Documents

To avoid the 577 * 3538 comparisons of every requirement from one document against every requirement from the other, we developed a technique based on a prototype study explained in Appendix A. The technique involves string searching to locate other requirements with identical keywords, supported by a table-of-contents comparison.

5. Identify Matches

Once a potential match was detected, we evaluated the meaning of the keywords within each requirement. We inferred meanings from their locations in each requirements

document, the meaning of neighboring requirements, and the subset-superset relationships of the two requirements. Fully matching requirements had sufficient semantic similarity to convince the analyst that each meaning of a keyword of one requirement is contained fully in the explicit or inferred meanings of the matching requirement. Our uni-directional matching process did not guarantee a one-to-one match between the two requirements, but did identify all AMPS requirements fully contained in both documents. FIGURE 4-2 contains examples of completely matched requirements.

AMPS 3.1.01 AMPS must integrate the applicable DII modules and/or standards into its own structure.

JMPS-090-03100 JMPS shall provide initial Defense Information Infrastructure Common Operating Environment (DII COE) compliance for Windows NT of at least Level-6 and a goal of evolution to compliance at Level-7.

AMPS 3.1.2.1.5 FLOPPY DISK DRIVE. The AMPS shall contain the necessary hardware and S/W drivers required to be able to read and write files to high density 3.5" FDs via a standard FDD.

JMPS-081-00050 JMPS shall provide the capability to support the physical interfaces that are supported by the Windows NT 4.0 operating system.

FIGURE 4-2. Fully Matching Requirement Examples

To identify all JMPS requirements fully contained in both documents, we would repeat the process in the reverse direction.

E. IDENTIFY OVERLAPPING REQUIREMENTS

Any matches that do not satisfy the requirements for a complete match are tagged as partial matches. Partial overlapping matches are requirements that do not have the same scope as the base requirement, as shown in Figure 4-3.

<p>AMPS 3.1.3.2.2 NEW CODE. For compatibility and supportability, all new production code developed as AMPS S/W shall be written in the ANSI Standard C or C++ programming language.</p> <p>JMPS-016-01030 JMPS shall provide API descriptions for GI&S tools using a language based on open standards, including Object Management Group (OMG) Interface Definition Language (IDL).</p>
--

FIGURE 4-3. Overlapping Requirement Example

F. IDENTIFY UNMATCHED REQUIREMENTS

Unmatched requirements represent functionality found only in one system. For the AMPS and JMPS documents, unmatched requirements often indicate external interfaces to other unique Army systems, transactions, data representations, and devices. As shown in Figure 4-4.

AMPS 3.1.2.1.10.1.1 Radio Communication Networks. In order to support transmission of information via the TCIM to IDM or ATHS equipped aircraft or other AMPS, AMPS must provide the means to set, modify, and delete the following Radio Network Parameters: Net Definition.

FIGURE 4-4. Unmatched Requirement Example

G. VALIDATE THE REQUIREMENT MATCHES AND OVERLAPS WITH THE STAKEHOLDERS

The manual matching output was validated by two domain analysts not associated with the matching project. Corrections were made based on their input[MATH00]. About 25% of the AMPS requirements were modified by the domain analysts. The primary reasons for modification were in the decomposition of requirements and in the inclusion of additional partial matching JMPS requirements.

H. INTEGRATION OF REQUIREMENTS MATCHES AND OVERLAPS INTO A DOMAIN MODEL

The manual matching process used a relational database (Microsoft Access) to store requirements and maintain all matching information. FIGURE 4-5 shows a screen capture of the forms used to display the matching data.

Microsoft Access

File Edit View Insert Format Records Tools Window Help

AMPS/JMPS Requirement Trace

Requirements No# 3.1.3.2.2

Requirement Specification

NEW CODE. For compatibility and supportability, all new production code developed as AMPS S/W shall be written in the ANSI Standard C or C++ programming language.

Core(C)/Aircraft(A)/Unit(U)
NA(N)/Unknown(?)
Requirement
C

Hardware(H)/Software(S)
NA(N)/Personnel(P)/Logistical(L)/D
ocumentation(D)/Unknown(?)
Security(Z)
LD

Version: Current(C)/Future(F)
NA(N)/Unknown(?)
Requirement
C

Evaluation Status:
Evaluated(X)/Unknown(?)
EvaluatedNoReq(N)
?

Notes

JMPS Req Spec #1 JMPS-016-01030

Requirements Specification	Time	Provider
JMPS shall provide API descriptions for GI&S tools using a language based on open standards, including Object Management Group (OMG) Interface Definition Language (IDL).^	V1	GTRI

Record: 1 of 1

JMPS Req Spec #2

Requirements Specification	Time	Provider
----------------------------	------	----------

Record: 1 of 1

JMPS Req Spec #3

JMPS Req Spec #4

JMPS Req Spec #5

Record: 211 of 577

Form View

Start Exploring - AMPS Req Work Microsoft Access Microsoft Word - AMPSAn...

11:13 PM

FIGURE 4-5. Manual Matching Database Display

THIS PAGE INTENTIONALLY LEFT BLANK

V. EVALUATION OF THE MANUAL MATCHING PROCESS

The manual matching process identified the AMPS SSS requirements satisfied by JMPS SSS requirements. This was the central question for the AMPS to JMPS migration. The manual-matching process selected the five best requirement matches from the JMPS document for each AMPS requirement. We used this to prototype an automated matching process, develop matches for comparison with our automated matching tool, and gain familiarity with the domain.

The manual matching process first identified 3538 JMPS requirements, 397 composite AMPS requirements, 577 AMPS requirements after partitioning, 1547 domain keywords, and 883 matching requirements. Of the 883 matches, 148 were one-to-one and 735 were many-to-one. We had reduced the 577 AMPS requirements to 467 after a second pass to remove requirements without domain-relevant content. Figure 5-1 shows an example.

3.1.1.1.2 Reserved.

3.1.1.2.1 Commercial Power Mode. The AMPS shall be capable of continuous operation using power from domestic or foreign commercial utility sources of 110-220 volts alternating current (AC).

FIGURE 5-1. Removed AMPS Requirement Examples

The average time to manually evaluate a requirement varied from five to thirty minutes. Matches between requirements in similar sections of each document were easier to evaluate due to many domain terms nearby; these

helped in identifying matches with larger syntactic and conceptual differences.

The evaluation time decreased as we gained experience with the requirements and the document partitioning. The total time required to match all 467 AMPS requirements to the 3538 requirements in the JMPS document was 110.5 hours (6420 minutes). The average time to match a requirement to all requirements in the other document was fourteen minutes. This time included the search time required to find the five best matching requirements. We selected domain-relevant keywords during the knowledge-acquisition process over a two-month period but estimate that it takes about five minutes per requirement with moderate domain expertise.

A major limitation of this manual matching process was the lack of time available to find and evaluate all matching pairs. Due to our self-imposed limitation of 3-5 keywords per requirement, we primarily evaluated matching requirements within the domains captured by the keywords. We estimate that a manual matching process that considered all keywords in a requirement would take three to four times longer than the process used in our study.

In addition, we limited the time spent in the manual matching process by selecting only the five best matching JMPS requirements for each AMPS requirement. Our definition of a complete match allowed for combination of up to five

JMPS requirements to fully satisfy an AMPS requirement. When insufficient JMPS requirements existed to satisfy an AMPS requirement, we classified them as partially matching requirements. When no matching JMPS requirements were found, we classified the AMPS requirement as unmatched.

Appendix B contains a spreadsheet breakdown of the results of the manual matching process.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. AUTOMATED MATCHING METHODOLOGY

The purpose of the automated tool is to reduce the time an analyst spends in searching for complete and partially matching requirements from two requirements documents. The tool does not make match decisions.

A. DEVELOPMENT GOALS

Based on our manual matching experience, we developed an automated matching tool (AMT) using keyword matching. Since searching for matches consumed most of the time in the manual study, our goals for the tool were to:

- reduce the number of evaluated requirements;
- reduce irrelevant words from consideration in each requirement;
- ignore as much as possible, the differences due to grammar, tense, punctuation, and capitalization;
- provide a similarity value for rank ordering of requirements;
- group requirements by locality for matching;
- reduce bookkeeping tasks but make no matching decisions;
- provide a solution that is platform-independent; and
- minimize the customization needed for a given requirements document.

We assume that the analyst is familiar with the concept of matching requirements using unique requirements numbers. The user will provide a key to uniquely identify each requirement. We assume the user can perform basic tasks with text files (i.e. open, close, save to a directory). We assume that the user can access a Java-capable computer with the ability to read, write, and save text files.

B. LANGUAGE

The AMT is written in Java and uses the Java Virtual Machine (JVM) and Java Developers Kit 1.3 libraries for its run-time execution. We selected Java as our development language to support platform independence, be consistent with the increasing use of Java in software development, and permit future web-based enhancements. Java also supports object-oriented design that enforces information hiding, a desirable feature for large software engineering tasks.

C. PARSING PREPARATION

Converting the requirements documents into a format for parsing was straightforward. We saved each requirements document as a text file, removed all non-requirements text within each document (i.e. table of contents, introduction, glossaries, and appendices), and analyzed each document to identify the coding for requirements numbers. The AMPS system used numbers separated by a period like 3.2.4.1.2

that began with "3."; the JMPS system used alphanumeric strings like JMPS-001-00000 that began with "JMPS-0". The String Tokenizer method from Java was used to divide each document up into individual word tokens. Requirement numbers were markers for beginnings of the requirements.

D. WORD MATCHING

Determining a match between two requirements involved word comparisons using Java. Java compares strings lexicographically [SUN99] with the compareTo() method.

We compared strings as part of a rough measure of the similarity between two requirements. We can use this to rank order matched pairs. The similarity is computed in these steps, an improvement on the classic inner-product formula [SALT88].

- determine the number of occurrences(N) of a word(i) within a given requirement(j) in document(a)

$$(Na_{ij})$$

- divide that value by the total number of occurrences of the word within the requirements document

$$(Na_{ij}) / (Na_i)$$

- multiply this result by the same result computed for a requirement in a second document(b)

$$\left[\frac{(Na_{ij})}{(Na_i)} \right] * \left[\frac{(Nb_{ij})}{(Nb_i)} \right]$$

- Sum the set of results for all words (Z) in the initial requirement

$$\sum_{i=1}^Z \left[\left[\frac{(Na_{ij})}{(Na_i)} \right] * \left[\frac{(Nb_{ij})}{(Nb_i)} \right] \right]$$

This total should be normalized by the relative frequency of words in each document. Requirements matching on infrequently occurring words should have a higher similarity value. In addition, requirements with more words should not have higher totals than shorter requirements with fewer words and the same degree of correlation. The normalization factor is:

$$\sqrt{\sum_{i=1}^Z \left[\left(\frac{(Na_{ij})}{(Na_i)} \right)^2 \right]} * \sqrt{\sum_{i=1}^Z \left[\left(\frac{(Nb_{ij})}{(Nb_i)} \right)^2 \right]}$$

So the similarity between requirement j_a and j_b is:

$$S(j_a, j_b) = \frac{\sum_{i=1}^Z \left[\left[\frac{(Na_{ij})}{(Na_i)} \right] * \left[\frac{(Nb_{ij})}{(Nb_i)} \right] \right]}{\sqrt{\sum_{i=1}^Z \left[\left(\frac{(Na_{ij})}{(Na_i)} \right)^2 \right]} * \sqrt{\sum_{i=1}^Z \left[\left(\frac{(Nb_{ij})}{(Nb_i)} \right)^2 \right]}}$$

We calculated similarity values for every pair of requirements with at least one common word. We stored these results in a hashtable for recall and comparison with the manual matching results.

E. REFINEMENTS TO BASIC WORD MATCHING

The Stop Word Filter removes unhelpful words (e.g. "rather", "really", "require", "requirement") from the word list within each requirement. This reduces the number of word comparisons that the tool must make and reduces the false matches. It removes 613 words and phrases that occur commonly in the English language [ROWE99]. Most of the words came from MARIE-2 [ROWE99] but we added certain domain-irrelevant words that occur with a high frequency (i.e. "JMPS", "AMPS", single characters, single digit numbers, "requirement").

The Upper-Case Elimination Filter decreases the number of lexicographically different words with identical spelling but different capitalization. We used the `toLowerCase()` method from Java. But we excluded acronyms from this process, defining them as words in all capitals.

The Destemming Filter written by Rowe truncates the suffixes of words to derive their root forms using an algorithm adapted from Porter [PORT80]. Destemming also reduces the number of different words in the documents.

We determined that most effective way to filter was to apply in order the Stop Word Filter, The Upper-Case Elimination Filter, and the Destemming Filter. Removing stop words first allows proper nouns to be removed prior to conversion to lower case; destemming is the slowest filtering and so benefits from being last.

F. MACROREQUIREMENTS

Macrorequirements are groups of adjacent related requirements, a concept important in the manual matching study. Our study showed that similar sections contained semantic matches with varying degrees of lexicographical similarity. So we explored automatic aggregation of requirements into Macrorequirements. We used a word-count threshold to group requirements. The algorithm coalesced requirements until the threshold was crossed. We used the defined hierarchy of the requirements document to group requirements and the requirement numbers to identify leaf nodes. We grouped leaves with their parents until the threshold was exceeded. We then computed similarity values on Macrorequirements instead of individual requirements.

VII. EVALUATION OF THE AUTOMATED MATCHING PROCESS

We tested the different filters to determine their performance improvements against manual matching. Every pair found by automated matching that occurred in the 883 manual matches and exceeded the similarity threshold was considered a success. The output was plotted as a recall vs. precision curve. Recall measures the completeness of a search [INFO98]. We computed recall as the number of successful matches divided by the number of complete or partial matches identified in manual matching(883). Precision measures the signal-to-noise ratio [INFO98]. We computed as the total number of successful matches divided by the total number of matches that the tool rated with similarity exceeding the threshold. We tabulated the number of matched pairs with similarity values in given ranges. The ranges were bounded by values of 0.9, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, and 0.0001.

A. WORD MATCHING

Word Parsing Totals

Similarity Value	Precision	Recall	Intersection Count	Tool Matches
> 0.9	0	0	0	10
> 0.5	0.0854	0.0113	10	117
> 0.1	0.0205	0.0804	71	3461
> 0.05	0.0131	0.1223	108	8201
> 0.01	0.0065	0.2582	228	34834
> 0.005	0.0051	0.3193	282	55097
> 0.001	0.0026	0.4088	361	138406
> 0.0005	0.0019	0.4462	394	197243
> 0.0001	0.0011	0.556	491	436478
> 0.00005	0.0008	0.5968	527	588241
> 0.0	0.0005	0.7938	701	1223769

Word Parsing

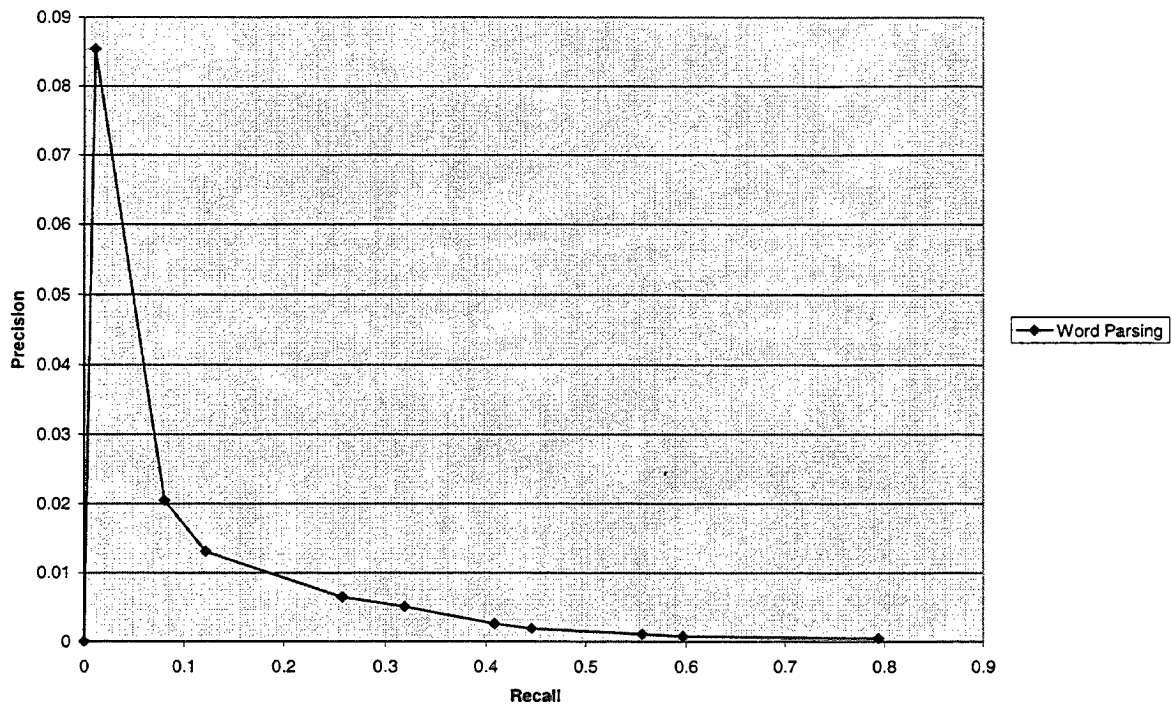


FIGURE 7-1. Recall/Precision Curve of Word Matching

B. COMBINED FILTER MATCHING

Combination Filter Totals Similarity Value	Precision	Recall	Intersection Count	Tool Matches
> 0.9	0.0714	0.0011	1	14
> 0.5	0.0477	0.0170	15	314
> 0.1	0.0186	0.0997	88	4731
> 0.05	0.0128	0.1427	126	9816
> 0.01	0.0067	0.2752	243	36535
> 0.005	0.0055	0.3477	307	56174
> 0.001	0.0036	0.5096	450	124125
> 0.0005	0.0030	0.5504	486	164144
> 0.0001	0.0019	0.6433	568	293771
> 0.00005	0.0016	0.6670	589	364310
> 0.0	0.0011	0.7508	663	629622

Combination Filter

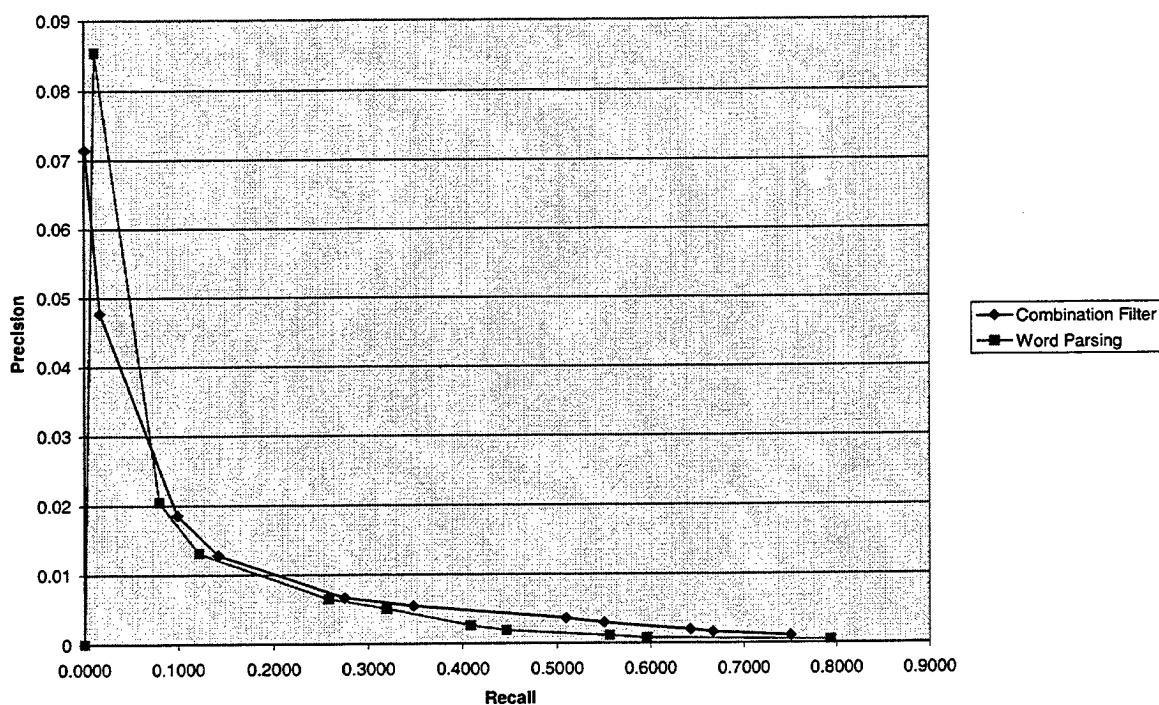


FIGURE 7-2. Recall/Precision Curve of Combined Filter Matching (Stop Word/Upper-Case Elimination/Destemmer)

C. MACROREQUIREMENT BUNDLED MATCHING

MacroRequirement - 100 word
bundles

Similarity Value	Precision	Recall	Intersection Count	Tool Matches	Manual Matches
> 0.9	0.0000	0.0000	0	0	401
> 0.5	0.0000	0.0000	0	0	401
> 0.1	0.0952	0.0200	8	84	401
> 0.05	0.1006	0.0823	33	328	401
> 0.01	0.0553	0.3641	146	2640	401
> 0.005	0.0434	0.5362	215	4956	401
> 0.001	0.0245	0.8354	335	13649	401
> 0.0005	0.0205	0.9077	364	17766	401
> 0.0001	0.0163	0.9850	395	24281	401
> 0.00005	0.0154	0.9875	396	25761	401
> 0.0	0.0142	1.0000	401	28149	401

MacroRequirements

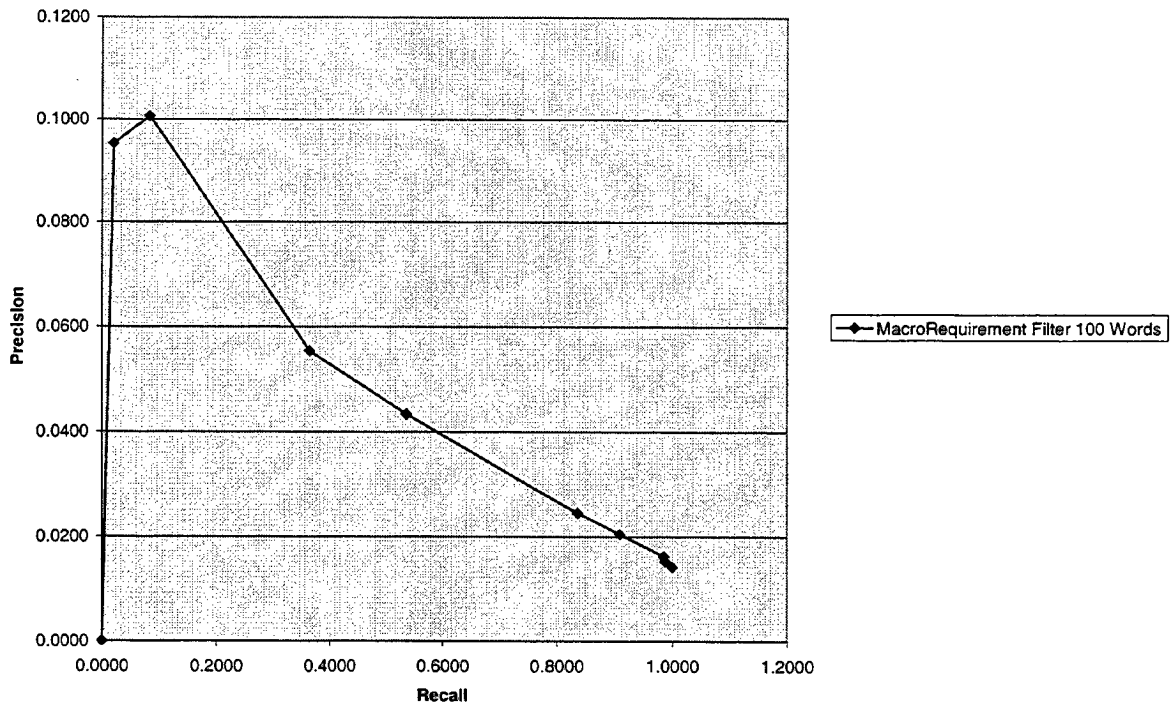


FIGURE 7-3. Recall/Precision Curve of MacroRequirement Bundled Matching

D. RESULTS

The total time needed to match AMPS with JMPS requirements with the automated tool is approximately one hour. Using the Combined Filter, the recall is 74% when considering all matches with at least one nontrivial word in common, with a precision of 0.0001. Even considering the low precision, in this case the tool has reduced the analyzable matches from 2,041,426 to 620,132 for a 70% reduction.

When using the Combined Filter, the tool rated 49 correct matches with the highest similarity value for any of the matches involving either of their requirements. This means that the best manual match choice was rated highest by the automated system for 8.5% of the requirements. Assuming that evaluation of manual matches will occur at 18474 per hour (2,041,426 matches / 110.5 hrs), the remaining manual matching will only take 33.6 hours. This yields a 69% improvement in the time required to analyze two documents for matching requirements.

The tool output is a list of matched pairs in order from highest to lowest similarity value. Each matched pair displays the original text from the associated requirements. The analyst inspects each match to determine if a complete or partial match exists and progresses through the list until a complete match is found, all partial matches are

identified or the list is exhausted. Figure 7-8 illustrates a potential matched pair.

AMPS-3-1-2-1-9-1
<u>Requirement Text</u> : MODEMS. The AMPS shall contain the necessary hardware and S/W drivers required to exchange files via a standard (i.e. Hayes compatible) modem.
JMPS-006-03220
<u>Requirement Text</u> : JMPS Data Communications shall support exchanges via modem on external telecommunications circuits including worldwide commercial communications lines, Defense Messaging System (DMS) circuits, and tactical communications circuits.
<u>Similarity Value</u> = 0.3037

FIGURE 7-8. Tool Matching Output

The analyst reads the two requirements and decides that it is a tentative partial match. They refer to the source documents, review the context of the two requirements, and make a final determination that it is a partial match because of the specificity of the hardware in the AMPS requirements.

Figure 7-3 shows an example of bundled matching with MacroRequirements.

VIII.CONCLUSIONS & FUTURE WORK

The manual matching study concluded that over 76% of the AMPS requirements were completely matched by the JMPS requirements. However, the manual matching process was restricted to consider only the best five requirements matched on the domain keywords, which may limit identification of partial matches for other requirements documents. Proximity of similar requirements when determining a match was a key factor in finding matches but was sometimes misleading.

Our experiences showed that care should be taken if the requirements documents contain many composite requirements. Domain analysts must develop skills or find domain experts who can assist in dividing the requirements into sub-domains for analysis. This cost may offset the benefit of the reuse effort.

Based on these results, we concluded that an automated tool using proximity factors, but not exclusively, could reduce the search time for potential matching requirements. Our experiences with an automated matching tool indicated that much time can be saved over current practices of manual matching for requirements reuse. Through an automated keyword-matching process that incorporates information-retrieval techniques such as destemming, stop-word lists,

and case-sensitivity adjustment, the analysis time can be reduced by 70%.

We also concluded that domain entities could be easily identified and converted to an XML text file ready for elaboration in the Spec formal language. We created supporting XML documents that contained the information needed to understand the context of the domain entity.

We provided an XML Document Type Definition(DTD) formal model that ensures each entity in the domain model conforms to the Spec BNF[BERZ89] and the language definitions in [BERZ91]. We selected Spec syntax for the XML DTD because of:

Support for inheritance: This simplified the descriptions of each element, supports standardization, and improves view integration on large domains.

Support for temporal logic (states): This permits checking the model to verify conformance to a specification.

Support for mathematical theorem-proving: This allows the domain description to be checked against axioms or inference rules.

User familiarity: This reduces the problems associated with the steep user learning curve as documented in [NEIL98].

The process of analyzing a requirements document using information-retrieval ideas in an automated process opens up

the possibility to capitalize on the wealth of domain knowledge in legacy systems considered for migration to next-generation systems. Converting these legacy-system requirements into axiomatic domain models can reduce cost and risk while improving time-to-market.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: MANUAL MATCHING PROTOTYPE

During the manual-matching research, a semi-formal study was conducted to select a strategy to match requirements. The following options were considered as variations when selecting a match for a given requirement:

1. Conduct a manual inspection of the matching document text using a provided index and the analyst's recall of the matching requirement's document organization and content.

2. Use the search feature of a word-processing package to find matches of keywords.

3. Combine the two techniques: Do an exhaustive search for keywords followed by a manual inspection of the relevant sections.

A semi-formal test was conducted with five requirements on each technique for speed. The results were the same for both techniques. The pure manual matching was conducted first followed by a partially-automated string search. The following set of requirements were extracted at random from the AMPS document:

"3.1.1.5.3 Corrective Actions. The system shall prompt the user on corrective actions required to either resolve the detected faults or to abort the actions."

"3.1.2.1.14 LARGE SCREEN DISPLAY. The AMPS shall contain the necessary hardware and S/W drivers required to be able to output its display screens to an external large

screen display of a size and resolution suitable for presentation to groups no smaller than 12 persons while simultaneously displaying on the built in screen for the operator."

"3.1.2.4.6 GLOBAL POSITIONING SYSTEM. The AMPS shall provide the capability to convert GPS Almanac data and waypoint data from mission route function format into the database formats required to load GPS systems identified in section 3.10.1."

"3.1.3.2.2 NEW CODE. For compatibility and supportability, all new production code developed as AMPS S/W shall be written in the ANSI Standard C or C++ programming language."

"3.2.2.5 ELECTRONIC FILE TRANSFER. The AMPS user shall be able to transfer data to any of the media supported by the interfaces specified in section 3.1.2.1 (e.g. DTC, HDD, FDD, MODD, etc.) and from-to any combination of these supported media which are appropriate to the data being transferred."

AMPS Requirement Number	3.1.1.5.3	3.1.2.1.14	3.1.2.4.6	3.1.3.2.2	3.2.2.5	Totals (min)
Base Document						
Document Partitioning	1	1	1	1	1	5
Section Review	3	7	3	2	5	20
Requirement Review	1	3	5	2	3	14
Match Document						
Document Partitioning	1	2	1	1	1	6
Section Review	15	7	10	10	8	50
Requirement Review	3	10	6	5	9	33
Number of Matching Requirement Numbers	NONE	1	2	1	1	5
Total Time / Requirement (Minutes)	24	30	26	21	27	128

FIGURE A-1. Manual Search Technique Results

AMPS Requirement Number	3.1.1.5.3	3.1.2.1.14	3.1.2.4.6	3.1.3.2.2	3.2.2.5	Totals (min)
Base Document						
Document Partitioning	1	1	1	1	1	5
Section Review	3	7	3	2	5	20
Requirement Review	1	3	5	2	3	14
Match Document						
Document Partitioning	1	2	1	1	1	6
Section Review	0	0	0	0	0	0
Requirement Review	3	5	5	3	4	20
Number of Matching Requirement Numbers	NONE	0	1	0	2	3
Total Time / Requirement (Minutes)	9	18	15	9	14	65

FIGURE A-2. String Search Matching Results

It appears the semi-automated keyword search reduced the time to identify potential requirement matches. However, its accuracy was definitely inferior to the manual inspection as it had a 40% less precise result. Based on this, the combination approach was adopted.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: MANUAL MATCHING RESULTS

AMPS/JMPS	No# AMPS Req	No# JMPS Matches	JMPS V1 Matches Only	JMPS Future Req	JMPS Mixed V1/Future Req	JMPS Patial Match
Requirements Matching Results						
Current AMPS Requirements						
- Non-aircraft Specific	35	27	15	12	0	1
- Aircraft Specific	70	31	10	9	12	10
- Mixed Act/Common	10	10	8	2	0	0
- Common	281	266	184	35	47	9
SubTotal	396	334	217	58	59	20
Future AMPS Requirements						
- Non-aircraft Specific	26	5	5	0	0	13
- Aircraft Specific	13	3	2	1	0	0
- Mixed Act/Common	0	0	0	0	0	0
- Common	32	27	23	1	3	2
SubTotal	71	35	30	2	3	15
Total Requirements	467	369	247	60	62	35
Percent JMPS to AMPS Match	79.01%					
Percent JMPS to V1 Match	52.89%					

FIGURE B-1. Manual Matching Study Results

The term "V1", refers to version 1 release of the JMPS software. Future refers support for future requirements not funded for development in JMPS V1.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: AUTOMATED TOOL SOURCE CODE

```
//Title:      DocObject
//Version:    1.0
//Author:     Eric Stierna
//Company:    NPS
//Description: This class holds the set of requirements contained in
//            an instance of a requirements document.

package parseproj;

import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DocObject {

    // Name of the document that contains the set of requirements
    String docTitle = "Document Name Not Provided";

    // This Hashtable contains the list of requirement objects
    // one requirement per hashed entry
    Hashtable reqObjectList = new Hashtable();

    // list of all words and the number of occurrences in the document
    // hashed on the word in the table
    Hashtable tokenList = new Hashtable();

    // list of requirements associated with a given word hashed on the
    // word they are associated with
    Hashtable tokenToReqList = new Hashtable();

    public DocObject(String docTitleString)
    {
        docTitle = docTitleString;
    }

    public String getDocTitle()
    {
        return(docTitle);
    }

    public Hashtable getReqObjectList()
    {
        return(reqObjectList);
    }

    public ReqObject getReqObject(String inputReqNum)
    {
        return((ReqObject)reqObjectList.get(inputReqNum));
    }

    // this method adds an ReqObject as an entry in the hash
    // table, keyed on the requirement number
    public void addReqObject(String reqNum, ReqObject newReq)
    {
        reqObjectList.put(reqNum, newReq);
    }

    // this method adds a word from the requirement Object to a
    // hashtable of words in the document
    public void addToTokenList(String newToken)
    {
        if (tokenList.containsKey(newToken))
        {
            RequirementWord tempWordToken =
```

```

        (RequirementWord) tokenList.get(newToken);

        tempWordToken.incrementWordCount();
    }
    else
    {
        RequirementWord newTokenRecord =
            new RequirementWord(newToken);

        tokenList.put(newToken, newTokenRecord);
    }
}

// This method allows the user to look-up the requirements
// associated with each word in the document
public void addToTokenToReqList(String newToken, String reqNumber)
{
    if (tokenToReqList.containsKey(newToken))
    {
        ReqToken tempReqToken =
            (ReqToken) tokenToReqList.get(newToken);

        tempReqToken.addReqNumber(reqNumber);
    }
    else
    {
        ReqToken newReqTokenRecord =
            new ReqToken(newToken, reqNumber);

        tokenToReqList.put(newToken, newReqTokenRecord);
    }
}

public Hashtable getTokenList()
{
    return(tokenList);
}

public Hashtable getTokenToReqList()
{
    return(tokenToReqList);
}

public Vector getReqListFromToken(String wordToken)
{
    ReqToken myToken = (ReqToken) tokenToReqList.get(wordToken);

    return(myToken.getReqList());
}

public int getTokenCount(String tokenQuery)
{
    int tempCount = 0;

    if (tokenList.containsKey(tokenQuery))
    {
        RequirementWord tempWordToken = (RequirementWord)
            tokenList.get(tokenQuery);
        tempCount = tempWordToken.getWordCount();
    }
    return(tempCount);
}

public int getReqCount()
{
    return(reqObjectList.size());
}

public int getWordCount()
{

```

```

        return(tokenList.size());
    }

    // This method is used to compute the normalization factor for
    // similarity computations
    public void updateReqPValForDocument()
    {
        ReqObject tempReqObject;

        // compute the similarity value denominator for one
        // document
        Enumeration ReqEnum = reqObjectList.elements();

        // loop through each requirement and set the reqPVal
        while (ReqEnum.hasMoreElements())
        {
            tempReqObject = (ReqObject)ReqEnum.nextElement();

            // set the ReqPVal
            tempReqObject.setReqPVal(this);
        }
    }
} // end of DocObject file

//Title:         ReqObject
//Version:        1.0
//Author:         Eric Stierna
//Company:        NPS
//Description: This class holds the info contained in a requirement.

package parseproj;

import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This class holds the info contained in a requirement.
 * @author Eric Stierna
 */

public class ReqObject {

    // String container for the unique requirement number in a given
    // document
    String reqNumber = "Requirement Number Not Provided";

    // Hashtable contains RequirementWord objects that each contain
    // a unique word from the requirement and the number of occurrences
    // of the word in the requirement.
    Hashtable tokenList = new Hashtable();

    // This is equal to the sqrt of the sum of the wordPVals squared
    // for a given requirement. Each WordPVal is equal to the
    // number of occurrences of a word in a requirement divided by the
    // number of occurrences of the word in the document.
    double reqPVal = 0.0;

    // count of the number of words in the ReqObject
    // (includes duplicate words in the count)
    int reqWordCount = 0;

    /**
     * ReqObject constructor
     * @param reqNumString - the requirement number in string format
     */
    public ReqObject(String reqNumString)
    {

```

```

        reqNumber = reqNumString;
    }

    /**
     * ReqObject constructor
     * @param reqNumString - the requirement number in string format
     * @param stringList - list of word strings in a vector container
     */
    public ReqObject(String reqNumString, Vector wordList)
    {
        // set the req number attribute
        reqNumber = reqNumString;

        Enumeration tempEnumeration = wordList.elements();

        // loop through the enumeration adding each element to the
        // wordList hashtable using the addReqWord method
        while (tempEnumeration.hasMoreElements())
        {
            addReqWord((String) tempEnumeration.nextElement());
        }
    }

    /**
     * addReqWord method - adds word tokens to the reqObject Hashtable
     * @param newWord - the new word token
     */
    public void addReqWord(String newWord)
    {
        reqWordCount = reqWordCount + 1;

        // increment word count if the word already exists in the
        // hashtable
        if (tokenList.containsKey(newWord))
        {
            RequirementWord wordRecord = (RequirementWord)
                tokenList.get(newWord);
            wordRecord.incrementWordCount();
        }
        // else create a new object and add it to the hashtable
        else
        {
            RequirementWord newWordRecord =
                new RequirementWord(newWord);

            tokenList.put(newWord, newWordRecord);
        }
    }
    // end of addToken method

    /**
     * getReqNumber method - gives access to the requirement number
     * @return reqNumber <code>String</code> returns the reqNum
     */
    public String getReqNumber()
    {
        return(reqNumber);
    }

    /**
     * getReqPVal method - This method returns the double
     * containing the PVal for a given req in a document.
     * @return reqPVal <code>double</code> returns a double
     */
    public double getReqPVal()
    {
        return(reqPVal);
    }

    /**
     * getTokenList method - gives access to the tokenList
     * @return tokenList <code>Hashtable</code> returns the list

```

```

    * of tokens in a Hashtable.
    */
    public Hashtable getTokenList()
    {
        return(tokenList);
    }

    /**
     * getReqWordCount method - gives access to the reqWordCount
     * @return reqWordCount <code>int</code> returns the number
     * of words in the requirement.
     */
    public int getReqWordCount()
    {
        return(reqWordCount);
    }

    /**
     * getTokenOccurance method - This method returns the number of
     * occurrences of a word within a given requirement.
     * @param checkToken - word to use for the query
     * @return tempVal <code>int</code> returns the number of
     * occurrences of a word
     */
    public int getTokenOccurance(String checkToken)
    {
        int tempVal = 0;

        if (tokenList.containsKey(checkToken))
        {
            tempVal = ((RequirementWord)
                tokenList.get(checkToken)).getWordCount();
        }
        return(tempVal);
    }

    /**
     * setReqPVal method - This method allows a document to set
     * the reqPVal for its reqObjects. This method is called by the
     * doc object which passes it the document's tokenList which gives
     * access to the number of occurrences of a word in the document.
     * @param DocTokenList - Hashtable containing a look-up table for
     * word occurrences in a document
     */
    public void setReqPVal(DocObject owningDocument)
    {
        int tempCount = 0;
        int totalDocTokenCount = 0;
        double tempPVal = 0.0;
        double summationTotal = 0.0;

        // This enumeration gets the list of words that are in the
        // actual requirement object.
        Enumeration reqWordList = tokenList.elements();

        // while hashtable is not empty
        // iterate through the list of words
        while (reqWordList.hasMoreElements())
        {
            // get a word
            RequirementWord localWordRecord = (RequirementWord)
                reqWordList.nextElement();

            int tempDocCount = owningDocument.
                getTokenCount(localWordRecord.getReqWord());
            // Get wordRecord's word String and use the string to
            // hash into the DocTokenList hashtable to extract the
            // requirementWord which is then used to get the word

```

```

        // count
        if(tempDocCount != 0)
        {

            // extract the number of occurrences of the word
            // in the requirement
            tempCount = localWordRecord.getWordCount();

            // divide the no# occurrences of the word in the
            // req by the number of occurrences in the
            // document.
            tempPVal = ((double) tempCount) /
                ((double) tempDocCount);

            // raise the result to the second power
            tempPVal = Math.pow(tempPVal, 2.0);

            // add the result to a summation total
            summationTotal = summationTotal + tempPVal;
        }
        else
        {
            System.out.println
                ("Unaccounted word in Req Token List");
        }
    } // end while loop

    // set the reqPVal
    reqPVal = Math.sqrt(summationTotal);
} // end of setReqPVal() method

/**
 * getTokenOccurance method - This method computes the PVal for
 * each word in a reqObject. This method is called by the document
 * object which passes it the word count and a string to identify
 * the word.
 * @param tokenString - String containing the word
 * @param docTokenCount - int count of the occurrences in the doc
 * @return tempPVal <code>double</code> returns the PVal for the
 * token.
 */
public double getTokenPVal(String tokenString,
                           int docTokenCount)
{
    int tempCount = 0;
    double tempPVal = 0.0;

    if (tokenList.containsKey(tokenString))
    {
        RequirementWord wordRecord = (RequirementWord)
            tokenList.get(tokenString);

        // get the number of occurrences of the word in the req
        tempCount = wordRecord.getWordCount();

        if (docTokenCount != 0)
        {
            // compute the tokenPVal// compute the tokenPVal
            tempPVal = ((double)tempCount /
                (double)docTokenCount);
        }
    }
    return(tempPVal);
}

} // end of ReqObject Class

//Title:         RequirementWord

```

```

//Version:          1.0
//Author:           Eric Stierna
//Company:          NPS
//Description: This class contains one word(string), counter, and
//                boolean flag to indicate if the word is a stop word.

```

```

package parseproj;

import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RequirementWord {

    String wordString = "No Word Assigned";

    // The number of occurrences of a word in a requirement.
    int wordCount = 1;

    // by default it is not a stop word.
    boolean stopWord = false;

    public RequirementWord(String inputString)
    {
        wordString = inputString;
    }

    public RequirementWord(String inputString, boolean stopFlag)
    {
        wordString = inputString;

        if(stopFlag == true)
        {
            setStopWord();
        }
    }

    public RequirementWord(String inputString, int newWordCount)
    {
        wordString = inputString;
        wordCount = newWordCount;
    }

    public String getReqWord()
    {
        return(wordString);
    }

    public void incrementWordCount()
    {
        wordCount++;
    }

    public void addToWordCount(int incVal)
    {
        wordCount = wordCount + incVal;
    }

    public int getWordCount()
    {
        return(wordCount);
    }

    private void setStopWord()
    {
        stopWord = true;
    }

```

```

    }

    public boolean getStopWordStatus()
    {
        return(stopWord);
    }

} // end of RequirementWord Class

//Title:          ReqToken
//Version:         1.0
//Author:          Eric Stierna
//Company:         NPS
//Description: This class contains one word(string) and an associated
//              Vector of requirement numbers or macro numbers in
//              which the word occurs.
//              The class prevents duplicate entries of the same
//              requirement.

package parseproj;

import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ReqToken {

    String wordToken = "No Word Assigned";

    // list of requirements in which the word occurs.
    Vector reqList = new Vector();

    public ReqToken(String inputWordToken, String inputReqNum)
    {
        wordToken = inputWordToken;

        reqList.add(inputReqNum);

    } // end reqToken() constructor

    public String getWordToken()
    {
        return(wordToken);
    }

    public Vector getReqList()
    {
        return(reqList);
    }

    public void addReqNumber(String inputReqNum)
    {
        if (!reqList.contains(inputReqNum))
        {
            reqList.add(inputReqNum);
        }
    }

} // end of ReqToken Class

//Title:          ReqSimObject
//Version:         1.0
//Author:          Eric Stierna
//Company:         NPS
//Description: This class stores matching req info.
//              It contains two req numbers(strings) and a
//              double to indicate the similarity between the two req.
//              The first req number is the base and the second is the

```

```

//                                matching req.

package parseproj;

import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ReqSimObject {

    private String baseReqNumString = "No Word Assigned";
    private String matchReqNumString = "No Word Assigned";
    private double productReqPVal = 0.0;
    private double similarityVal = 0.0;
    private double summationVal = 0.0;

    // This comparator allows ReqSimObjects to be compared.
    // First the baseReqObjStrings are compared (low to hi)
    // Second the SimilarityValues are compared (hi to low)
    // Last the matchReqObjStrings are compared (low to hi)
    static final Comparator REQ_SIM_OBJ = new Comparator()
    {
        public int compare(Object o1, Object o2)
        {
            ReqSimObject r1 = (ReqSimObject) o1;
            ReqSimObject r2 = (ReqSimObject) o2;

            int newReqNumString = r1.getBaseReqNumString().
                compareTo(r2.getBaseReqNumString());
            if (newReqNumString != 0)
            {
                return newReqNumString;
            }
            else
            {
                return (r1.getMatchReqNumString().
                    compareTo(r2.getMatchReqNumString()));
            }
        }
    };

    // used to create sim objects when a comma seperated
    // list of matching requirements is already available.
    public ReqSimObject(String newBaseReqNumString,
                        String newMatchReqNumString)
    {
        baseReqNumString = newBaseReqNumString;
        matchReqNumString = newMatchReqNumString;
    }

    public ReqSimObject ( String newBaseReqNumString,
                        String newMatchReqNumString,
                        double newBaseReqPVal,
                        double newMatchReqPVal,
                        double newBaseTokenPVal,
                        double newMatchTokenPVal)
    {
        baseReqNumString = newBaseReqNumString;
        matchReqNumString = newMatchReqNumString;

        productReqPVal = newBaseReqPVal * newMatchReqPVal;

        updateSimPVal(newBaseTokenPVal, newMatchTokenPVal);
    }

    public String getBaseReqNumString()
    {
        return(baseReqNumString);
    }
}

```

```

    public String getMatchReqNumString()
    {
        return(matchReqNumString);
    }

    public double getSimilarityVal()
    {
        return(similarityVal);
    }

    /**
     * updateSumPVal method - computes the summation of similarity
     * values corresponding to word matches between requirements
     * @param currentSimObj - existing ReqSimObject
     * @param newBaseTokenPVal - base document word PVal
     * @param newMatchTokenPVal - match document word PVal
     */
    public void updateSimPVal(double newBaseTokenPVal,
                              double newMatchTokenPVal)
    {
        summationVal = summationVal +
            (newBaseTokenPVal * newMatchTokenPVal);

        if (productReqPVal != 0)
        {
            similarityVal = summationVal / productReqPVal;
        }
        else
        {
            System.out.println("Attenpting to divide by zero");
        }
    }

    public String getKey()
    {
        return(baseReqNumString + matchReqNumString);
    }
} // end of ReqSimObject Class

//Title:           MatchObject
//Version:          1.0
//Author:           Eric Stierna
//Company:          NPS
//Description: This class creates a list of matching requirements and
//              their similarity value for two documents.

package parseproj;

import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MatchObject
{

    float capacity = (float)0.9;

    // This Hashtable contains the list of ReqSimObjects that each
    // contain an instance of a match between two requirements and
    // a similarity value
    Hashtable matchedReqList = new Hashtable(1000000, capacity);

    // this int is used in statistical analysis to determine the
    // number of matches between two sets of requirements.
    int matchCount = 0;

```

```

// constructor allows the user to set the similarity match factor
public MatchObject(DocObject baseDoc, DocObject matchDoc)
{
    // compute the similarity value denominator for the
    // base document
    baseDoc.updateReqPValForDocument();

    // compute the similarity value denominator for the
    // match document
    matchDoc.updateReqPValForDocument();

    determineMatch(baseDoc, matchDoc);
}

// constructor allows user to build a match object from a
// prematched list of requirements captured in a text file.
public MatchObject(String fileName, String reqDelimString)
{
    buildMatchObjectFromPreMatchedList(fileName,
        reqDelimString);
}

private void determineMatch(DocObject baseDoc,
    DocObject matchDoc)
{
    // Temp variables for casting and computations
    ReqObject tempReqObject;
    ReqObject tempReqObjectBase;
    ReqObject tempReqObjectMatch;

    RequirementWord tempWordTokenBase;

    double tempReqPValBase = 0.0;
    double tempTokenPValBase = 0.0;
    double tempReqPValMatch = 0.0;
    double tempTokenPValMatch = 0.0;
    double tempSimilarityVal = 0.0;

    String hashString = "NA";

    // declare a new enumeration to loop through all ReqObjects
    // in the base document
    Enumeration baseReqEnum = baseDoc.getReqObjectList().
        elements();

    // get list of ReqObjects from baseDoc DocObject
    // loop through the list of req objects matching each
    // requirement
    // with 0..* requirements from the matchDocument
    while (baseReqEnum.hasMoreElements())
    {
        // get a baseDoc ReqObject
        tempReqObjectBase =
            (ReqObject)baseReqEnum.nextElement();

        // get the reqPVal for the baseDoc ReqObject
        tempReqPValBase = tempReqObjectBase.getReqPVal();

        // declare an enumeration of the list of word tokens
        // that are contained in the requirement object
        Enumeration baseWordEnum = tempReqObjectBase.
            getTokenList().elements();

        // loop through the list of words in each ReqObject
        // computing the SimilarityVal between each matched
        // requirement
        // and storing it in a ReqSimObject. Store all

```

```

// ReqSimObjects
// in the Hashtable
while (baseWordEnum.hasMoreElements())
{
    // get a RequirementWord from a baseDoc
    // ReqObject
    tempWordTokenBase = (RequirementWord)
        baseWordEnum.nextElement();

    // only perform this block of code if there is a
    // match between the word and at least one req
    // in
    // the match document.
    if (matchDoc.getTokenToReqList().containsKey(
        tempWordTokenBase.getReqWord() ) )
    {
        // get the number of occurrences of the
        // word in the document and use the value // to get
        // the
        // tokenPVal for the word(baseDoc)
        tempTokenPValBase =
            tempReqObjectBase.getTokenPVal(
                tempWordTokenBase.getReqWord(),
                baseDoc.getTokenCount(
                    tempWordTokenBase.getReqWord() ) );

        // declare an enumeration of the list of
        // req that are related to that token.
        Enumeration newMatchReqEnum =
            (matchDoc.getReqListFromToken(
                tempWordTokenBase.getReqWord())).elements();

        // use the enumerated list of matching
        // requirements
        // to loop through the list of
        // requirements getting
        // each requirement object in turn.
        while (newMatchReqEnum.hasMoreElements())
        {
            // get a ReqObject from the matchDoc
            tempReqObjectMatch = (
                matchDoc.getReqObject(
                    (String)newMatchReqEnum.nextElement() ));

            // get the tokenPVal for the word in
            // the ReqObject
            // uses the word from the outer
            // while loop to do the
            // look-up
            tempTokenPValMatch =
                tempReqObjectMatch.
                getTokenPVal(tempWordTokenBase.getReqWord(),
                    matchDoc.getTokenCount(tempWordTokenBase.
                        getReqWord()));

            // check the hashTable if a matching
            // ReqSimObject
            // is found then execute the sumPVal
            // method
            // hash on the concatenation of the
            // two
            // requirement number strings
            if (matchedReqList.containsKey(
                tempReqObjectBase.getReqNumber() +
                tempReqObjectMatch.getReqNumber() ))
            {
                // use the reqNumbers as a key
                // to hash into
                // the ReqSimObject Hashtable
                // and get the

```

```

        // matching ReqSimObject,
        // execute updateSimPVal
        // on the ReqSimObject by
        // passing in the two
        // tokenPVals
        ((ReqSimObject)matchedReqList.
get(tempReqObjectBase.getReqNumber()
+
tempReqObjectMatch.getReqNumber()))
updateSimPVal(tempTokenPValBase,
tempTokenPValMatch);

    }
    // else get the reqPVal for the
    // ReqObject, execute
    // the updateSimVal method and
    // create a newReqSimObject with the //
    gathered info else
    {
        // get the the reqPVal for the //
        ReqObject
        tempReqPValMatch =
        tempReqObjectMatch.
        getReqPVal();

        // create a newReqSimObject
        // using all six
        // parameters: 2 req numbers,
        // 2 reqPVal,
        ReqSimObject newReqSimObject =
        new ReqSimObject(
        tempReqObjectBase.getReqNumber(),
        tempReqObjectMatch.getReqNumber(),
        tempReqPValBase,
        tempReqPValMatch,
        tempTokenPValBase,
        tempTokenPValMatch);

        // store the ReqSimObject in
        // the matchedReqList
        // Hashtable
        matchedReqList.put(tempReqObjectBase.
        getReqNumber() +
        tempReqObjectMatch.getReqNumber(),
        newReqSimObject);

        matchCount++;
    } // end else

    // exit the while loop when no more
    // ReqObjects remain in the list
    }
} // end of if strucutre check for base req word

// with no more matching req from the list in matchDoc
// exit the loop when no more words in the ReqObject
// remain in
// the list
}
// exit the loop when no more ReqObjects in the
// DocObject remain in
// the list
}

} // end of determineMatch() method

public Hashtable getMatchedReqList()
{
    return(matchedReqList);
}

```

```

// outputs matching req list
// assumes the matching method has allready been performed
public void matchListOutput(String outputFileName)
{
    try
    {
        // open the file to write to
        BufferedWriter outputText = new BufferedWriter(
            new FileWriter(outputFileName));

        // create a TreeSet Object and use the ReqSimObject
        // comparator to sort all added contents
        TreeSet sortedMatchList =
            new TreeSet(ReqSimObject.REQ_SIM_OBJ);

        // add all matching requirements to the TreeSort
        sortedMatchList.addAll(
            matchedReqList.values());

        // create an iterator to use to output an ordered
        // list
        Iterator myIterator = sortedMatchList.iterator();

        // temp var to hold the req number
        String reqNumHdr = null;

        // loop until all elements in the TreeSort have been
        // visited.
        while (myIterator.hasNext())
        {
            // extract a ReqSimObject from the iterator
            ReqSimObject tempObject =
                (ReqSimObject) myIterator.next();

            // write to output file
            outputText.write(tempObject.getBaseReqNumString());

            outputText.write("\t" +
                tempObject.getMatchReqNumString() +
                "\t" + tempObject.getSimilarityVal());

            outputText.newLine();
        }
        outputText.close();
    } // end of try

    catch(FileNotFoundException e)
    {
    }

    catch(IOException e)
    {
    }

} // end of matchListOutput

// extracts requirements from a text file and creates a match
// object. The text file must contain a list of requirements,
// one per line. The method finds the first delimiter that
// uniquely identifies a requirement and matches each token on
// subsequent lines until the end of the file or another
// delimiter is encountered
private void buildMatchObjectFromPreMatchedList
    (String fileName,
     String reqDelimString)
{
    try // Lvl 1

```

```

{
    // read in the fileName
    BufferedReader inputText = new BufferedReader(
        new FileReader(fileName));

    // declarations/initializations
    String s1 = "Dummy String";
    String baseReqNum = "NA";
    String delimString = "\n";

    // loop until the readline method sets s1 to null
    while (s1 != null)
    {
        try // Lvl 2
        {
            // read in a line from the file
            s1 = inputText.readLine();

            // tokenize the line
            StringTokenizer st =
                new StringTokenizer(s1,
                    delimString);

            // loop while more tokens exist
            while(st.hasMoreTokens())
            {
                // get a copy of the current token
                String newWord = st.nextToken();
                // if the token already exists in
                // the hash table
                if ( newWord.regionMatches(true,
                    0,
                    reqDelimString,
                    0,
                    reqDelimString.length()) )
                {
                    baseReqNum = newWord;
                }
                else
                {
                    // create a new Requirement
                    // word object
                    ReqSimObject newSimObject =
                        New
                        ReqSimObject(baseReqNum,
                            newWord);
                    // enter it into the hash
                    // table
                    matchedReqList.put(baseReqNum
                        + newWord, newSimObject);
                    matchCount++;
                }
            }
            // end while(hasMoreTokens)
        } // end of try Lvl 2

        catch(IOException e)
        {
            s1 = null;
        }
        catch(NullPointerException e)
        {
        }

        // end of while(s1 != null)
        // close the input file
        inputText.close();
    } // end of try Lvl 1
    catch(FileNotFoundException e)
    {

```

```

        }
        catch(IOException e)
        {
        }
    } // end of buildMatchObjectFromPreMatchedList() method

    public int getMatchCount()
    {
        return(matchCount);
    }

} // end of matchObject file

//Title:      MacroRequirement
//Version:    1.0
//Author:     Eric Stierna
//Company:    NPS
//Description: This class is a container for a group of requirements
//            and their words.

package parseproj;

import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This class is a container for a group of requirements
 * • and their words.
 * @ author Eric Stierna
 */

public class MacroRequirement
{
    private String macroNumber = "DefaultMacroNumber";
    private Vector reqList = new Vector();
    private Hashtable wordList = new Hashtable();
    private int wordCount = 0;
    private double macroPVal = 0.0;
    private int macroLevel = 1;
    private boolean isBundled = false;

    /**
     * MacroRequirement constructor
     * @param macroNumber - the unique number to id the macro req
     */
    public MacroRequirement(String newMacroNumber)
    {
        macroNumber = newMacroNumber;
    }

    /**
     * MacroRequirement constructor
     * @param macroNumber - the unique number to id the macro req
     * @param newReq - the first requirement object is added to the
     * macro requirement.
     */
    public MacroRequirement(String newMacroNumber, ReqObject newReq)
    {
        macroNumber = newMacroNumber;
        addRequirement(newReq);
    }

    /**
     * MacroRequirement constructor
     * @param macroNumber - the unique number to id the macro req
     * @param oldMacro - the old macro requirement to be bundled with
     * the new MacroRequirement

```

```

    */
    public MacroRequirement(String newMacroNumber,
        MacroRequirement oldMacro)
    {
        macroNumber = newMacroNumber;
        addMacroRequirement(oldMacro);
    }

    public void addRequirement(ReqObject newReq)
    {
        setMacroLevel(newReq);
        reqList.add(newReq.getReqNumber());
        updateWordList(newReq.getTokenList());
    }

    public void addMacroRequirement(MacroRequirement oldMacro)
    {
        this.macroLevel = oldMacro.getMacroLevel();
        Enumeration newReqList = oldMacro.getReqList().elements();
        while( newReqList.hasMoreElements())
        {
            this.reqList.add(newReqList.nextElement());
        }
        updateWordList(oldMacro.getWordList());
    }

    private void updateWordList(Hashtable reqWordList)
    {
        Enumeration wordEnum = reqWordList.elements();
        while (wordEnum.hasMoreElements())
        {
            RequirementWord tempReqWord = (RequirementWord)
                wordEnum.nextElement();

            if (wordList.containsKey(tempReqWord.getReqWord()))
            {
                RequirementWord oldWord = (RequirementWord)
                    wordList.get(tempReqWord.getReqWord());

                oldWord.addToWordCount(tempReqWord.getWordCount());
            }
            else
            {
                RequirementWord newWord =
                    new RequirementWord(tempReqWord.getReqWord(),
                        tempReqWord.getWordCount());

                wordList.put(newWord.getReqWord(), newWord);
            }
        }
    } // end of updateWordList()

    public String getMacroNumber()
    {
        return(macroNumber);
    }

    public Vector getReqList()
    {
        return(reqList);
    }

    public Hashtable getWordList()
    {
        return(wordList);
    }

    public int getMacroWordCount()
    {
        return(wordCount);
    }

```

```

    }

    public double getMacroPVal()
    {
        return(macroPVal);
    }

    /**
     * setReqPVal method - This method allows a MasterMacro to set
     * the macroPVal for its MacroRequirements. This method is called
     * by the MasterMacroRequirement
     * object which passes it the document's tokenList to give access to the
     * number of occurrences of a word in the document.
     * @param DocTokenList - Hashtable containing a look-up table for word
     * occurrences in a document
     */
    public void setReqPVal(DocObject owningDocument)
    {
        int tempCount = 0;
        int totalDocTokenCount = 0;
        double tempPVal = 0.0;
        double summationTotal = 0.0;

        // This enumeration gets the list of words that are in the
        // actual requirement object.
        Enumeration reqWordList = wordList.elements();

        // while hashtable is not empty
        // iterate through the list of words
        while (reqWordList.hasMoreElements())
        {
            // get a word
            RequirementWord localWordRecord = (RequirementWord)
reqWordList.nextElement();

            int tempDocCount = owningDocument.
                getTokenCount(localWordRecord.
                    getReqWord());

            // Get wordRecord's word String and use the string to
            // hash into the DocTokenList hashtable to extract the
            // requirementWord which is then used to get the word
            // count
            if(tempDocCount != 0)
            {
                // extract the number of occurrences of the word in the
                // requirement
                tempCount = localWordRecord.getWordCount();

                // divide the no# occurrences of the word in the req by the
                // number of occurrences in the document.
                tempPVal = ((double) tempCount) / ((double) tempDocCount);

                // raise the result to the second power
                tempPVal = Math.pow(tempPVal, 2.0);

                // add the result to a summation total
                summationTotal = summationTotal + tempPVal;
            }
            else
            {
                System.out.println("Unaccounted word in Req Token List");
            }
        }

        // end while loop

        // set the reqPVal
        macroPVal = Math.sqrt(summationTotal);
    }

```

```

} // end of setReqPVal() method

/**
 * getTokenOccurance method - This method computes the PVal for each
 * word in a MacroRequirement. This method is called by the
 * MasterMacroRequirement
 * which passes it the word count and a string to identify the word
 * @param tokenString - String containing the word
 * @param docTokenCount - int count of the occurrences in the doc
 * @return tempPVal <code>double</code> returns the PVal for the token
 */
public double getTokenPVal(String tokenString,
                           int docTokenCount)
{
    int tempCount = 0;
    double tempPVal = 0.0;

    if (wordList.containsKey(tokenString))
    {
        RequirementWord wordRecord = (RequirementWord)
wordList.get(tokenString);

        // get the number of occurrences of the word in the req
        tempCount = wordRecord.getWordCount();

        if (docTokenCount != 0)
        {
            // compute the tokenPVal// compute the tokenPVal
            tempPVal = ((double)tempCount / (double)docTokenCount);
        }
        return(tempPVal);
    }

public int getMacroLevel()
{
    return(macroLevel);
}

public void setMacroLevel(ReqObject newReqObj)
{
    Vector levelList = new Vector();

    // convert the requirement number to a character array
    char[] stringToCharArray = newReqObj.getReqNumber().
toCharArray();

    // remove all non-digits from the char array and
    // insert each digit into a vector to hold the remainder of
    // the requirement number
    for(int i=0; i<stringToCharArray.length; i++)
    {
        Character newChar = new Character(stringToCharArray[i]);
        if (newChar.isDigit(stringToCharArray[i]))
        {
            newChar = new Character(stringToCharArray[i]);
            levelList.add(newChar);
        }
    }

    // loop control variable for trimming zeros off the end of
    // the JMPS req numbers
    boolean moreZeros = true;
    int vectorIndex = levelList.size() - 1;

    while (moreZeros == true)
    {
        Character myChar = (Character)
            levelList.elementAt(vectorIndex);
        if (myChar.charValue() == 0)

```

```

        {
            levelList.remove(vectorIndex);
            vectorIndex--;
        }
        else
        {
            moreZeros = false;
        }
    }

    macroLevel = levelList.size();
}

public void setMacroNumber(String bundledNumber)
{
    macroNumber = bundledNumber;
}

public boolean getIsBundled()
{
    return(isBundled);
}

public void setIsBundledTrue()
{
    isBundled = true;
}

} // end of MacroRequirement() Class

//Title:      MasterMacroRequirement
//Version:    1.0
//Author:     Eric Stierna
//Company:    NPS
//Description: This object contains the macroRequirements for a given
//            document.

package parseproj;

import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @author Eric Stierna
 */

public class MasterMacroRequirement
{
    private String macroName = "DefaultMasterMacroNumber";

    // Contains a look-up list of all MacroRequirements given the
    // MacroNumber
    private Hashtable macroList = new Hashtable();

    // Contains a look-up of all MacroRequirements that contain a
    // given Requirement. Given a requirement number string, the
    // hashtable returns a macroReqNumber String.
    private Hashtable reqToMacroList = new Hashtable();

    // Contains a look-up of all MacroRequirements that contain a
    // given word. Given a word string, the
    // hashtable returns a list of matching MacroRequirements.
    private Hashtable wordToMacroList = new Hashtable();

    // global counter used to define unique macroNumbers
    private int numberCount = 0;

```

```

        // Points to the current MacroRequirement
        private MacroRequirement currentMacro = new
        MacroRequirement("emptyMacro");

private Vector tempList = new Vector();

    /**
     * MasterMacroRequirement constructor
     */
    public MasterMacroRequirement(String newMacroName)
    {
        macroName = newMacroName;
    }

    public Hashtable getMacroList()
    {
        return(macroList);
    }

    public MacroRequirement getMacroReq(String macroReqNumber)
    {
        return((MacroRequirement)macroList.get(macroReqNumber));
    }
    // The boolean helps this method deal with the last requirements
    public void add(ReqObject newReqObj, boolean lastReqObj)
    {
        MacroRequirement nextMacro = new MacroRequirement("MacroReq-" +
        macroName + "-" + numberCount, newReqObj);
        numberCount++;

        // when peer is found
        if(currentMacro.getMacroLevel() == nextMacro.getMacroLevel())
        {
            System.out.println("Peer");

            // add the MacroRequirement to a temporary list
            tempList.add(nextMacro);

            currentMacro = nextMacro;

            if(lastReqObj)
            {
                if(!tempList.isEmpty())
                {
                    System.out.println("track-subordinates");

                    //add the entire temp list to the current MacroRequirement
                    Enumeration tempEnum = tempList.elements();

                    // add all MacroReq in the temp list
                    while(tempEnum.hasMoreElements())
                    {
                        // create a new MacroReq to hold the list of Requirements
                        MacroRequirement newMacro = new MacroRequirement("MacroReq-" +
                        macroName + "-" + numberCount);
                        numberCount++;

                        newMacro.addMacroRequirement(
                            (MacroRequirement)tempEnum.nextElement() );

                        // add the new MacroRequirement to the macroList HashTable
                        macroList.put(newMacro.getMacroNumber(), newMacro);

                        //add the words to the WordToMacro Look-up hashtable
                        addWordsFromMacro(newMacro);

                        // add the requirements from the new MacroRequirement to the
                        // requirement to Macro look-up table
                        addToReqToMacroList(newMacro);
                    }
                }
            }
        }
    }

```

```

        System.out.println("Test" + newMacro.getMacroNumber());
    }
}
}
// when subordinate is found
else if (currentMacro.getMacroLevel() < nextMacro.getMacroLevel())
{
    if(lastReqObj)
    {
        if(!tempList.isEmpty())
        {
            System.out.println("track-subordinates");

            //add the entire temp list to the current MacroRequirement
            Enumeration tempEnum = tempList.elements();

            // add all MacroReq in the temp list
            while(tempEnum.hasMoreElements())
            {
                // create a new MacroReq to hold the list of Requirements
                MacroRequirement newMacro = new MacroRequirement("MacroReq-" +
                    macroName + "-" + numberCount);
                numberCount++;

                newMacro.addMacroRequirement(
                    (MacroRequirement)tempEnum.nextElement() );

                // add the new MacroRequirement to the macroList HashTable
                macroList.put(newMacro.getMacroNumber(), newMacro);

                //add the words to the WordToMacro Look-up hashtable
                addWordsFromMacro(newMacro);

                // add the requirements from the new MacroRequirement to the
                // requirement to Macro look-up table
                addToReqToMacroList(newMacro);

                System.out.println("Test" + newMacro.getMacroNumber());
            }
        }
    }

    System.out.println("Subordinate");

    // clear the list - we've found a more distant leaf
    tempList.clear();

    // add the subordinate MacroRequirement to a temporary list
    tempList.add(nextMacro);

    currentMacro = nextMacro;
}
// when a superordinate is found
else if (currentMacro.getMacroLevel() > nextMacro.getMacroLevel())
{
    if(!tempList.isEmpty())
    {
        System.out.println("Super-Ordinate");

        //add the entire temp list to the current MacroRequirement
        Enumeration tempEnum = tempList.elements();

        // create a new MacroReq to hold the list of Requirements
        MacroRequirement newMacro = new MacroRequirement("MacroReq-" +
            macroName + "-" + numberCount);
        numberCount++;

        // add all MacroReq in the temp list
        while(tempEnum.hasMoreElements())

```

```

        {
            newMacro.addMacroRequirement(
                (MacroRequirement)tempEnum.nextElement() );
        }

        // add the new MacroRequirement to the macroList HashTable
        macroList.put(newMacro.getMacroNumber(), newMacro);

        //add the words to the WordToMacro Look-up hashtable
        addWordsFromMacro(newMacro);

        // add the requirements from the new MacroRequirement to the
        // requirement to Macro look-up table
        addToReqToMacroList(newMacro);

        System.out.println("Test" + newMacro.getMacroNumber());

        // clear the tempList so that we don't backtrack
        tempList.clear();

        // set pointers
        currentMacro = nextMacro;
    }
}

} // end of add() method

public void outputMasterData()
{
    Enumeration tempEnum = macroList.elements();
    while (tempEnum.hasMoreElements())
    {
        System.out.println("Req is:" + ((MacroRequirement)tempEnum.
            nextElement()).getMacroNumber());
    }
}

public void updateMacroPVals(DocObject myDoc)
{
    MacroRequirement tempReqObject;

    // compute the similarity value denominator for one
    // MasterMacroRequirement
    Enumeration ReqEnum = macroList.elements();

    // loop through each requirement and set the reqPVal
    while (ReqEnum.hasMoreElements())
    {
        tempReqObject = (MacroRequirement)ReqEnum.nextElement();

        // set the ReqPVal
        tempReqObject.setReqPVal(myDoc);
    }
}

// This method allows the words from an existing requirement
// to be added to a MacroRequirement
public void addWordToMacroList(String newMacroNumber,
                                ReqObject newReq)
{
    Enumeration reqListEnum = newReq.getTokenList().elements();

    while (reqListEnum.hasMoreElements())
    {
        RequirementWord tempReqWord = (RequirementWord)
            reqListEnum.nextElement();

        if (wordToMacroList.containsKey(tempReqWord.getReqWord()))
        {

```

```

        ReqToken tempReqToken =
        (ReqToken)wordToMacroList.get(tempReqWord.getReqWord());

        tempReqToken.addReqNumber(newMacroNumber);
    }
    else
    {
        ReqToken newReqTokenRecord =
            new ReqToken(tempReqWord.getReqWord(),
newMacroNumber);

        wordToMacroList.put(tempReqWord.getReqWord(),
            newReqTokenRecord);
    }
}

// This method allows the words from an existing MacroRequirement
// to be added to the MasterMacroList
private void addWordsFromMacro(MacroRequirement newTempMacro)
{
    // create an enumeration of the new MacroRequirement's WordList
    Enumeration wordListEnum = newTempMacro.getWordList().elements();

    // while more words remain
    while (wordListEnum.hasMoreElements())
    {
        RequirementWord tempReqWord = (RequirementWord)
            wordListEnum.nextElement();

        // check to see if a look-up exists in the hash table
        if (wordToMacroList.containsKey(tempReqWord.getReqWord()))
        {
            ReqToken tempReqToken =
            (ReqToken)wordToMacroList.get(tempReqWord.getReqWord());

            // Assign the macro to the word
            tempReqToken.addReqNumber(newTempMacro.getMacroNumber());
        }
        // if not create a new entry
        else
        {
            ReqToken newReqTokenRecord =
                new ReqToken(tempReqWord.getReqWord(), newTempMacro.
getMacroNumber());

            wordToMacroList.put(tempReqWord.getReqWord(),
                newReqTokenRecord);
        }
    }
}

public Vector getWordToMacroList(String wordKey)
{
    if (wordToMacroList.containsKey(wordKey))
    {
        return(((ReqToken)wordToMacroList.get(wordKey)).
getReqList());
    }
    else
    {
        return(null);
    }
}

private void addToReqToMacroList(MacroRequirement tempMacro)
{
    Enumeration reqEnum = tempMacro.getReqList().elements();

```

```

        while (reqEnum.hasMoreElements())
        {
            reqToMacroList.put((String)reqEnum.nextElement(),
                               tempMacro.getMacroNumber());
        }
    }

    // This method adds all requirements from a macroReq to a
    // bundled requirement
    private void addReqsFromMacro(MacroRequirement newTempMacro)
    {
        Enumeration reqListEnum = newTempMacro.getReqList().elements();

        while (reqListEnum.hasMoreElements())
        {
            reqToMacroList.put((String)reqListEnum.nextElement(),
                               newTempMacro.getMacroNumber());
        }
    }

    public String getMacroNumberFromReqToMacroList(String reqNum)
    {
        return((String)reqToMacroList.get(reqNum));
    }
} // end of MasterMacroRequirement Class

//Title:           Word Parsing
//Version:          1.0
//Author:           Eric Stierna
//Company:          NPS
//Description: This class parses a text file into a hash table and
//              stores all instances of unique words from the document
//              along with the number of occurrences of each word.
//              Additional capabilities can be enabled to allow it to //
//              interact with the Wordnet 1.6 db and return word sense

package parseproj;

import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import java.awt.font.*;
import java.awt.geom.*;
import javax.swing.border.*;

public class WordParseMacroReq {

    public WordParseMacroReq()
    {
    }

    // tokenizes a preprocessed text file (one word per line)
    // into a hash table and sets a boolean flag to indicate if it
    // is a stop word (stop word = don't output)

    static private void fileReadAndParse(String fileName,
                                           Hashtable wordTable,
                                           boolean stopFlag)
    {
        try // Lvl 1
        {
            // read in the fileName
            BufferedReader inputText = new BufferedReader(
                new FileReader(fileName));

            // declarations/initializations
            String delimString = "\n";

```

```

        String s1 = "NA";

        // loop until the readline method sets s1 to null
        while (s1 != null)
        {
            try // Lvl 2
            {
                // read in a line from the file
                s1 = inputText.readLine();

                // tokenize the line
                StringTokenizer st =
                    new StringTokenizer(s1, delimString);

                // loop while more tokens exist in tokenizer
                while(st.hasMoreTokens())
                {
                    // get a token from the tokenizer
                    String s3 = st.nextToken();

                    // if the token already exists in the hash table
                    if (wordTable.containsKey(s3))
                    {
                        // get a copy of the hashed object
                        Object s2 = wordTable.get(s3);
                        if (s2 instanceof RequirementWord)
                        {
                            // cast object as a RequirementWord
                            RequirementWord wordRecord =

                                object
                                    (RequirementWord) s2;

                                wordRecord.incrementWordCount();
                        }
                        else
                        {
                            // create a new Requirement word
                            RequirementWord newWordRecord =
                                new RequirementWord(s3,

                                object
                                    stopFlag);

                            // enter it into the hash table
                            wordTable.put(s3, newWordRecord);
                        }
                    } // end while(hasMoreTokens)
                } // end of try Lvl 2

                catch(IOException e)
                {
                    s1 = null;
                }
                catch(NullPointerException e)
                {
                }

            } // end of while(s1 != null)

            // close the input file
            inputText.close();
        } // end of try Lvl 1
        catch(FileNotFoundException e)
        {
        }

        catch(IOException e)
        {
        }
    } // end of fileReadAndParse method

    // builds a list of the original requirements while converting

```

```

// the requirement numbers to the proper format. The requirements
// are stored in a hashtable for look-up with the req number as
// the keys.
static private void buildUnalteredReqList(String fileName,
                                           Hashtable
unalteredReqList,
                                           String
adjReqNumString,
                                           char[] reqNumPunc,
                                           String
reqDelimString)
{
    try // Lvl 1
    {
        // read in the fileName
        BufferedReader inputText = new BufferedReader(
            new FileReader(fileName));

        // declarations/initializations
        String delimString = "\t\n\r\f";
        String s1 = "Loop until null";

        // stores requirement tokens till saved
        Vector tokenList = new Vector();

        // flag indicates that the first requirement
        // has been encountered
        boolean firstReq = true;

        // holds requirement number while the requirement
        // text is stored in the vector
        String reqNum = "NA";

        // loop until the readline method sets s1 to null
        while (s1 != null)
        {
            try // Lvl 2
            {
                // read in a line from the file
                s1 = inputText.readLine();

                // tokenize the line - true indicates that all
                // delimiters should be returned as tokens
                StringTokenizer st =
                    new StringTokenizer(s1, delimString, true);

                // loop while more tokens exist in tokenizer
                while(st.hasMoreTokens())
                {
                    // get a token from the tokenizer
                    String s3 = st.nextToken();

                    // checks each token to determine if the
                    // reqDelimString occurs in the token
                    // starting with the first index
                    if (s3.regionMatches(true,
0,
                                           reqDelimString,
0,
                                           reqDelimString.length()) )
                    {
                        // if the string occurs then
                        // adds appendstring to token
                        s3 = adjReqNumString + s3;

                        // changes the punctuation in a req
number
                        s3.replace(reqNumPunc[0], reqNumPunc[1]);

```

```

encountered

tokenList.elements();
the req
reqNum;
(enumTokenList.hasMoreElements())
collectionString +
enumTokenList.nextElement();

hashtable
key
unaltered Req
collectionString);

// first req number has been
if (firstReq == true)
{
    reqNum = s3;
    firstReq = false;
}
else
{
    Enumeration enumTokenList =

    // declare a string to hold
    // text
    String collectionString =

    while
    {
        collectionString =
            " " + (String)

    }
    // fill the unalteredReqList
    // reqNum is the req number
    // collectionString is the
    unalteredReqList.put(reqNum,

    // prep for next requirement.
    reqNum = s3;
    tokenList.clear();
}
}
else
{
    tokenList.add(s3);
}
} // end while(st.hasMoreTokens())

} // end of try Lvl 2

catch(IOException e)
{
    s1 = null;
}
catch(NullPointerException e)
{
}

} // end of while(s1 != null)

// handles the last requirement
if (!tokenList.isEmpty())
{
    Enumeration enumTokenList = tokenList.elements();
    // declare a string to hold the req
    // text
    String collectionString = reqNum;

    while (enumTokenList.hasMoreElements())
    {

```

```

        collectionString = collectionString +
            " " + (String)
                enumTokenList.nextElement();
    }

    // fill the unalteredReqList hashtable
    // reqNum is the req number key
    // collectionString is the unaltered Req
    unalteredReqList.put(reqNum, collectionString);
}

// close the input file
    inputText.close();
} // end of try Lvl 1
catch(FileNotFoundException e)
{
}
catch(IOException e)
{
}
} // end of buildUnalteredReqList method

// modifies the source document by converting the document into
// one word per line, removing the delimiters and removing all
// stop words.
static private void preProcess(String inFileName,
                                String outFileName,
                                String delimString,
                                Hashtable wordTable,
                                int caseSelect)
{
    try // Lvl 1
    {
        // read in the fileName
        BufferedReader inputText = new BufferedReader(
            new FileReader(inFileName));

        // open the output file buffer stream
        BufferedWriter outputText = new BufferedWriter(
            new FileWriter(outFileName));

        // declarations/initializations
        String s1 = "Dummy String";
        String tokenTest;

        // loop until the readline method sets s1 to null
        while (s1 != null)
        {
            try // Lvl 2
            {
                s1 = inputText.readLine();

                if (caseSelect == 1)
                {
                    // tokenizes the file based on the string of
                    // passed into the method. true parameter
                    // delimiters to be returned as tokens
                    StringTokenizer st =
                        new StringTokenizer(s1, delimString);

                    while(st.hasMoreTokens())
                    {
                        tokenTest = st.nextToken();

                        if
                        {
                            tokenTest =

delimiters
causes all
(!tokenTest.equals(tokenTest.toUpperCase()))
tokenTest.toLowerCase();

```

```

        }
        // removes all stop word tokens
        if
        {
            // writes reamining tokens to
            outputText.write(tokenTest);
            outputText.newLine();
        }
        } // end while(hasMoreTokens)
    }
    else
    {
        // tokenizes the file based on the string of
        // passed into the method. true parameter
        // delimiters to be returned as tokens
        StringTokenizer st =
            new StringTokenizer(s1, delimString);

        while(st.hasMoreTokens())
        {
            tokenTest = st.nextToken();

            // removes all stop word tokens
            if
            {
                // writes reamining tokens to
                outputText.write(tokenTest);
                outputText.newLine();
            }
        } // end while(hasMoreTokens)
    }
    } // end of try Lvl 2

    catch(IOException e)
    {
        s1 = null;
    }
    catch(NullPointerException e)
    {
    }
    catch(NoSuchElementException e)
    {
    }

    } // end of while(s1 != null)

    // close the input file
    inputText.close();
    outputText.close();
    } // end of try Lvl 1
    catch(FileNotFoundException e)
    {
    }

    catch(IOException e)
    {
    }
    } // end of preProcess method

    // adds unique header to each req
    static private void appendPreProcess(String inFileName,
                                         String
                                         outFileName,

```

```

searchString,
appendString)
{
    try // Lvl 1
    {
        // read in the fileName
        BufferedReader inputText = new BufferedReader(
            new FileReader(inFileName));

        BufferedWriter outputText = new BufferedWriter(
            new FileWriter(outFileName));

        // declarations/initializations
        String s1 = "Dummy String";
        String tokenTest;

        // loop until the readline method sets s1 to null
        while (s1 != null)
        {
            try // Lvl 2
            {
                s1 = inputText.readLine();
                StringTokenizer st =
                    new StringTokenizer(s1);

                while(st.hasMoreTokens())
                {
                    tokenTest = st.nextToken();

                    // checks each token to determine if the
                    // occurs in the token starting with the
                    if ( tokenTest.regionMatches(true,

searchString
first index
0,
searchString,
0,

                                searchString.length()) )
                    {
                        // if the string occurs then
                        // adds appendString to token
                        tokenTest = appendString + tokenTest;
                        // customized bit of code because the

delimiter string
in these two chars in
req num

                        // does not remove hyphens
                        // ** need to add capability to pass
                        // a char array.
                        // changes all periods to dashes in

                        outputText.write(tokenTest.replace('.', '-'));
                        outputText.newLine();
                    }
                    else
                    {
                        outputText.write(tokenTest);
                        outputText.newLine();
                    }
                } // end while(hasMoreTokens)
            } // end of try Lvl 2

            catch(IOException e)
            {
                s1 = null;
            }
        }
    }
}

```

```

        catch(NullPointerException e)
        {
        }
        catch(NoSuchElementException e)
        {
        }

    } // end of while(s1 != null)

    // close the input file
    inputText.close();
    outputText.close();
} // end of try Lvl 1
catch(FileNotFoundException e)
{
}
catch(IOException e)
{
}
} // end of appendPreProcess method

// ReqPreProcessor method divides a req document into requirement
// objects.

// ***Note the inFileName File must begin with a req number.
// Each requirements must begin with common delimiter.
// use the appendPreProcess method to do this

// The method takes three strings and a document object(container):
// String inFileName - Name of the source file (designed to work with a text
//                      document)
// String delimString - character(s) used to identify tokens
// *note: could be made more robust by passing a boolean to indicate
//         if the delim is discarded. Currently it is not discarded.
// String reqDelimString - string used to identify the start of a req
static public void reqPreProcess(String inFileName,
                                String delimString,
                                String reqDelimString,
                                DocObject newDocument,
                                MasterMacroRequirement newMasterMacroReq,
                                boolean oneWayFlag)
{
    try // Lvl 1
    {
        // read in the fileName
        BufferedReader inputText = new BufferedReader(
            new FileReader(inFileName));

        // declarations/initializations
        String s1 = "Dummy String";
        String tokenTest;
        String reqNum = "Req Number Not Set";

        boolean firstReq = true;

        Vector tempTokenList = new Vector();

        // loop until the readLine method sets s1 to null
        while (s1 != null)
        {
            try // Lvl 2
            {
                s1 = inputText.readLine();
                StringTokenizer st =
                    new StringTokenizer(s1, delimString);

                while(st.hasMoreTokens())
                {
                    tokenTest = st.nextToken();

```

```

// checks for the start of a new requirement
if ( tokenTest.regionMatches(true,

0,
reqDelimString,
0,

reqDelimString.length()) )
{
// when not the first req number
// a req object
if (firstReq == false)
{
// create req object with
ReqObject newRequirement =
new ReqObject(reqNum,
tempTokenList);
// add req to document

newDocument.addReqObject(reqNum,
newRequirement);

newMasterMacroReq.add(newRequirement, false);

// capture the new req number
reqNum = tokenTest;
// clear the vector for the

// of tokens
tempTokenList.clear();

}
// capture req number
// set flag to start token capture
else
{
reqNum = tokenTest;
firstReq = false;
}
}
else
{
if (firstReq == false)
{
// add tokens to the temp list
// stores token in a vector of
// for creation of each new
tempTokenList.add(tokenTest);

if (oneWayFlag)
{
// add token and req
// documentmaster

which
strings
ReqObject

number to
tokenToReqList

newDocument.addToTokenToReqList(
tokenTest,
reqNum);
}
}
}
}

```

```

master word                                     // add token to document
                                                // token list

newDocument.addToTokenList(tokenTest);
    }
    } // end while(hasMoreTokens)
    st = null;
} // end of try Lvl 2
catch(IOException e)
{
    s1 = null;
}
catch(NullPointerException e)
{
}
catch(NoSuchElementException e)
{
}

} // end of while(s1 != null)
// handles the last requirement
if (!tempTokenList.isEmpty())
{
    // create req object with vector list
    ReqObject newRequirement1 =
        new ReqObject(reqNum, tempTokenList);
    // add req to document
    newDocument.addReqObject(reqNum, newRequirement1);
    newMasterMacroReq.add(newRequirement1,true);
    tempTokenList.clear();
}

// close the input file
    inputText.close();
} // end of try Lvl 1
catch(FileNotFoundException e)
{
}
catch(IOException e)
{
}
} // end of reqPreProcess method

/* // Special Processor method to provide the sense of each
// word in a "\r" delimited list.
static private void senseProcess(String inFileName,
                                String outFileName)
{
    try // Lvl 1
    {
        // read in the fileNames
        BufferedReader inputText = new BufferedReader(
            new FileReader(inFileName));

        BufferedWriter outputText = new BufferedWriter(
            new FileWriter(outFileName));

        // declarations/initializations
        String s1 = "Dummy String";
        String s2;
        String delimString = "\r";

        // loop until the readline method sets s1 to null
        while (s1 != null)
        {
            try // Lvl 2
            {
                s1 = inputText.readLine();
            }
        }
    }
}

```

```

        //tokenize based on the return at the end of the line
        StringTokenizer st =
            new StringTokenizer(s1, delimString);

        while(st.hasMoreTokens())
        {
            s2 = st.nextToken();
            outputText.write(s2);
            outputText.newLine();
            outputText.newLine();
            getSenseOfWord(s2, outputText);
            outputText.newLine();
            outputText.newLine();
        } // end while(hasMoreTokens)
    } // end of try Lvl 2

    catch(IOException e)
    {
        s1 = null;
    }
    catch(NullPointerException e)
    {
    }
    catch(NoSuchElementException e)
    {
    }

    } // end of while(s1 != null)

    // close the input file
    inputText.close();
    outputText.close();
} // end of try Lvl 1
catch(FileNotFoundException e)
{
}
catch(IOException e)
{
}

} // end of specPreProcess method
*/
/*
// outputs a list of hashed requirements records
static private void wordListOutput(String fileName,
                                     Hashtable
wordTable)
{
    try
    {
        BufferedWriter outputText = new BufferedWriter(
            new FileWriter(fileName));

        RequirementWord wordRecord;

        Enumeration wordList = wordTable.elements();

        while ( wordList.hasMoreElements() )
        {
            wordRecord = (RequirementWord) wordList.nextElement();
            if( !wordRecord.getStopWordStatus() )
            {
                String output = wordRecord.getReqWord();
                String tokenCount = wordRecord.getWordCount().toString();
                outputText.write(output + "," + "\t" + "\t");
                outputText.write(tokenCount);
                outputText.newLine();
            }
        } // end of while ( wordList.hasMoreElements() )
        outputText.close();
    } // end of try

    catch(FileNotFoundException e)

```

```

    {
    }

    catch(IOException e)
    {
    }
} // end of wordListOutput
*/
/*
// outputs matching req list
// assumes the matching method has allready been performed
// receives a list of requirement objects and an outputFileName
static public void reqListOutput(String outputFileName,
                                Vector
reqList)
{
    try
    {
        BufferedWriter outputText = new BufferedWriter(
            new FileWriter(outputFileName));

        ReqObject reqRecord;
        Hashtable reqMatchList;
        RequirementWord tempWord;
        Enumeration totalReqList = reqList.elements();
        Enumeration enumMatchList;
        Object s2;

        while ( totalReqList.hasMoreElements() )
        {
            // get a req from the list
            reqRecord = (ReqObject) totalReqList.nextElement();

            // create an enumeration of the matching req
            enumMatchList = reqRecord.getMatchList().elements();

            // output requirement number
            outputText.write(reqRecord.getReqNumber());
            outputText.newLine();

            while (enumMatchList.hasMoreElements())
            {
                s2 = enumMatchList.nextElement();
                if (s2 instanceof RequirementWord)
                {
                    tempWord = (RequirementWord)s2;
                    outputText.write("      " +

tempWord.getReqWord()
                                + "," + " " +

tempWord.getWordCount());
                                outputText.newLine();
                }
            }

            } // end of while ( totalReqList.hasMoreElements() )
        outputText.close();
    } // end of try

    catch(FileNotFoundException e)
    {
    }

    catch(IOException e)
    {
    }
} // end of reqListOutput
*/
/*
// This method wraps a simple method developed by Oliver Steele
// Copyright 1998 by Oliver Steele.

// The wrapper allows the method to take a given word and output

```

```

// the sense into the BufferedWriter.

// Changes include a generic string in lieu of the example word used
// to compute senses, changing the output from the console to an
// output file using BufferedWriter, and adding a try/catch to handle
// situations where the word has no sense with output feedback to the
// BufferedWriter.

// Changes made by Eric Stierna, Naval Postgraduate School
// Aug 2000

static private void getSenseOfWord(String sensedWord,
                                   BufferedWriter
outputText)
{
    try
    {
        DictionaryDatabase dictionary = new FileBackedDictionary();
        IndexWord word = dictionary.lookupIndexWord(POS.NOUN, sensedWord);

        try
        {
            Synset[] senses = word.getSenses();

            int taggedCount = word.getTaggedSenseCount();

            outputText.write("The " + word.getPOS().getLabel() + " " + word.getLemma()
+ " has " + senses.length + " sense" + (senses.length == 1 ? "" : "s") + " ");
            outputText.write("(");

            if (taggedCount == 0)
            {
                outputText.write("no senses from tagged texts");
            }
            else
            {
                outputText.write("first " + taggedCount + " from tagged texts");
            }

            outputText.write(")\n\n");

            for (int i = 0; i < senses.length; ++i) {
                Synset sense = senses[i];
                outputText.write(" " + (i + 1) + ". " + sense.getLongDescription());
            }
        }
        catch(Exception e)
        {
            outputText.write("No sense found in db for " + sensedWord + ".");
        }

    }
    catch(IOException e)
    {
        System.out.println("IOException!!!");
    }

} // end of getsense method
*/

// outputs matching req list
// assumes the matching method has allready been performed
static public void resultComparisonOutput(MatchObject manMatchObject,
                                           MatchObject
autoMatchObject,
                                           String outFileName,
                                           String
outReqFileName)
{
    try

```

```

{
    BufferedWriter outputText = new BufferedWriter(
        new FileWriter(outFileName));

    BufferedWriter outputMatchList = new BufferedWriter(
        new FileWriter(outReqFileName));

    // get the Hashtable, then create an
    // iterator to step through each manual req
    Enumeration autoEnum = autoMatchObject.

getMatchedReqList().elements();

    // intersection of matches divided by the manual matches
    double matchPrecision = 0.0;

    // intersection of matches divided by the manual matches
    double matchRecall = 0.0;

    int simArray[] = new int[11];
    int matchArray[] = new int[11];
    double simIncrement[] = { 0.00005, 0.00005, 0.0001, 0.0005, 0.001,
        0.005, 0.01, 0.05, 0.1, 0.5, 0.9};

    // loop till all manual matches have been evaluated
    // against the auto matches.
    while (autoEnum.hasMoreElements())
    {
        ReqSimObject tempObject =
            (ReqSimObject) autoEnum.nextElement();

        if (manMatchObject.getMatchedReqList().
            containsKey(tempObject.getKey()))
        {
            if (tempObject.getSimilarityVal() > simIncrement[10])
            {
                ++matchArray[10];
            }
            if (tempObject.getSimilarityVal() > simIncrement[9])
            {
                ++matchArray[9];
            }
            if (tempObject.getSimilarityVal() > simIncrement[8])
            {
                ++matchArray[8];
            }
            if (tempObject.getSimilarityVal() > simIncrement[7])
            {
                ++matchArray[7];
            }
            if (tempObject.getSimilarityVal() > simIncrement[6])
            {
                ++matchArray[6];
            }
            if (tempObject.getSimilarityVal() > simIncrement[5])
            {
                ++matchArray[5];
            }
            if (tempObject.getSimilarityVal() > simIncrement[4])
            {
                ++matchArray[4];
            }
            if (tempObject.getSimilarityVal() > simIncrement[3])
            {
                ++matchArray[3];
            }
            if (tempObject.getSimilarityVal() > simIncrement[2])
            {

```

```

        ++matchArray[2];
    }
    if (tempObject.getSimilarityVal() > simIncrement[1])
    {
        ++matchArray[1];
    }
    if (tempObject.getSimilarityVal() <= simIncrement[0])
    {
        ++matchArray[0];
    }
    outputMatchList.write(tempObject.getBaseReqNumString() +
        "," + tempObject.getMatchReqNumString() + "," +
        tempObject.getSimilarityVal());
    outputMatchList.newLine();
}
else
{
    if (tempObject.getSimilarityVal() > simIncrement[10])
    {
        ++simArray[10];
    }
    if (tempObject.getSimilarityVal() > simIncrement[9])
    {
        ++simArray[9];
    }
    if (tempObject.getSimilarityVal() > simIncrement[8])
    {
        ++simArray[8];
    }
    if (tempObject.getSimilarityVal() > simIncrement[7])
    {
        ++simArray[7];
    }
    if (tempObject.getSimilarityVal() > simIncrement[6])
    {
        ++simArray[6];
    }
    if (tempObject.getSimilarityVal() > simIncrement[5])
    {
        ++simArray[5];
    }
    if (tempObject.getSimilarityVal() > simIncrement[4])
    {
        ++simArray[4];
    }
    if (tempObject.getSimilarityVal() > simIncrement[3])
    {
        ++simArray[3];
    }
    if (tempObject.getSimilarityVal() > simIncrement[2])
    {
        ++simArray[2];
    }
    if (tempObject.getSimilarityVal() > simIncrement[1])
    {
        ++simArray[1];
    }
    if (tempObject.getSimilarityVal() <= simIncrement[0])
    {
        ++simArray[0];
    }
}
}
int i = 10;
while(i>=0)
{
    if (manMatchObject.getMatchCount() != 0)
    {
        matchRecall = (double)matchArray[i]/

```

```

(double)manMatchObject.getMatchCount();
    }
    if ((simArray[i] + matchArray[i]) != 0)
    {
        matchPrecision = (double)matchArray[i]/
            (double)(simArray[i] +
matchArray[i]);
    }
    System.out.println();
    System.out.println("Similiarity Value: " + simIncrement[i]);
    System.out.println("Total Manual Matches = "
        +
manMatchObject.getMatchCount());
    System.out.println("Total Matches Found by Tool(Base Case)= "
        + (simArray[i] +
matchArray[i]));
    System.out.println("Intersection Count = "
        + matchArray[i]);
    System.out.println("Precision = " + matchPrecision);
    System.out.println("Recall = " + matchRecall);

    outputText.write(matchPrecision + "," + matchRecall + "," +
        matchArray[i] + "," + (simArray[i] + matchArray[i]));
    outputText.newLine();

    i--;
}
outputText.close();
outputMatchList.close();
} // end of try
catch(FileNotFoundException e)
{
}

catch(IOException e)
{
}
} // end of resultComparisonOutput

static public void macroResultComparisonOutput
    (MacroMatchObject
manMatchObject,
    MacroMatchObject
autoMatchObject,
    String outFileName,
    String
outReqFileName)
{
    try
    {
        BufferedWriter outputText = new BufferedWriter(
            new FileWriter(outFileName));

        BufferedWriter outputMatchList = new BufferedWriter(
            new FileWriter(outReqFileName));

        // get the Hashtable, then create an
        // iterator to step through each manual req
        Enumeration autoEnum = autoMatchObject.
getMatchedReqList().elements();

        // intersection of matches divided by the manual matches
        double matchPrecision = 0.0;

        // intersection of matches divided by the manual matches
        double matchRecall = 0.0;

```

```

int simArray[] = new int[11];
int matchArray[] = new int[11];
double simIncrement[] = { 0.00005, 0.0001, 0.0005, 0.001,
    0.005, 0.01, 0.05, 0.1, 0.5, 0.9};

// loop till all manual matches have been evaluated
// against the auto matches.
while (autoEnum.hasMoreElements())
{
    ReqSimObject tempObject =
        (ReqSimObject) autoEnum.nextElement();

    if (manMatchObject.getMatchedReqList().
        containsKey(tempObject.getKey()))
    {
        if (tempObject.getSimilarityVal() > simIncrement[10])
        {
            ++matchArray[10];
        }
        if (tempObject.getSimilarityVal() > simIncrement[9])
        {
            ++matchArray[9];
        }
        if (tempObject.getSimilarityVal() > simIncrement[8])
        {
            ++matchArray[8];
        }
        if (tempObject.getSimilarityVal() > simIncrement[7])
        {
            ++matchArray[7];
        }
        if (tempObject.getSimilarityVal() > simIncrement[6])
        {
            ++matchArray[6];
        }
        if (tempObject.getSimilarityVal() > simIncrement[5])
        {
            ++matchArray[5];
        }
        if (tempObject.getSimilarityVal() > simIncrement[4])
        {
            ++matchArray[4];
        }
        if (tempObject.getSimilarityVal() > simIncrement[3])
        {
            ++matchArray[3];
        }
        if (tempObject.getSimilarityVal() > simIncrement[2])
        {
            ++matchArray[2];
        }
        if (tempObject.getSimilarityVal() > simIncrement[1])
        {
            ++matchArray[1];
        }
        if (tempObject.getSimilarityVal() <= simIncrement[0])
        {
            ++matchArray[0];
        }
        outputMatchList.write(tempObject.getBaseReqNumString() +
            "," + tempObject.getMatchReqNumString() + "," +
            tempObject.getSimilarityVal());
        outputMatchList.newLine();
    }
    else
    {
        if (tempObject.getSimilarityVal() > simIncrement[10])
        {
            ++simArray[10];
        }
    }
}

```

```

    }
    if (tempObject.getSimilarityVal() > simIncrement[9])
    {
        ++simArray[9];
    }
    if (tempObject.getSimilarityVal() > simIncrement[8])
    {
        ++simArray[8];
    }
    if (tempObject.getSimilarityVal() > simIncrement[7])
    {
        ++simArray[7];
    }
    if (tempObject.getSimilarityVal() > simIncrement[6])
    {
        ++simArray[6];
    }
    if (tempObject.getSimilarityVal() > simIncrement[5])
    {
        ++simArray[5];
    }
    if (tempObject.getSimilarityVal() > simIncrement[4])
    {
        ++simArray[4];
    }
    if (tempObject.getSimilarityVal() > simIncrement[3])
    {
        ++simArray[3];
    }
    if (tempObject.getSimilarityVal() > simIncrement[2])
    {
        ++simArray[2];
    }
    if (tempObject.getSimilarityVal() > simIncrement[1])
    {
        ++simArray[1];
    }
    if (tempObject.getSimilarityVal() <= simIncrement[0])
    {
        ++simArray[0];
    }
}
}
int i = 10;
while(i>=0)
{
    if (manMatchObject.getMatchCount() != 0)
    {
        matchRecall = (double)matchArray[i]/
(double)manMatchObject.getMatchCount();
    }
    if ((simArray[i] + matchArray[i]) != 0)
    {
        matchPrecision = (double)matchArray[i]/
(double)(simArray[i] +
matchArray[i]);
    }
    System.out.println();
    System.out.println("Similiarity Value: " + simIncrement[i]);
    System.out.println("Total Manual Matches = "
+
manMatchObject.getMatchCount());
    System.out.println("Total Matches Found by Tool (Base Case)= "
+ (simArray[i] +
matchArray[i]));
    System.out.println("Intersection Count = "
+ matchArray[i]);
    System.out.println("Precision = " + matchPrecision);
    System.out.println("Recall = " + matchRecall);
}

```

```

        outputText.write(matchPrecision + "," + matchRecall + "," +
            matchArray[i] + "," + (simArray[i] + matchArray[i]) +
            "," + manMatchObject.getMatchCount());
        outputText.newLine();

        i--;
    }
    outputText.close();
    outputMatchList.close();
} // end of try
catch(FileNotFoundException e)
{
}

catch(IOException e)
{
}
} // end of macroResultComparisonOutput

// this method takes a file containing a list of words that must
// be destemmed and returns a destemmed list of words.
static public void destemPreProcess(String newDestemWordFile,
                                    String validWordFile,
                                    String newDestemOutputFile)
{
    try // Lvl 1
    {
        // read in the fileName
        BufferedReader inputText = new BufferedReader(
            new FileReader(newDestemWordFile));

        // open the output file buffer stream
        BufferedWriter outputText = new BufferedWriter(
            new FileWriter(newDestemOutputFile));

        // declarations/initializations
        String s1 = "Dummy String";
        String delimString = "\n";

        Destem newDestemObject = new Destem();
        HashSet hs = new HashSet();
        newDestemObject.hashKnownWords(validWordFile, hs);

        // loop until the readline method sets s1 to null
        while (s1 != null)
        {
            try // Lvl 2
            {
                // read in a line from the file
                s1 = inputText.readLine();

                // tokenize the line
                StringTokenizer st =
                    new StringTokenizer(s1, delimString);

                // loop while more tokens exist
                while(st.hasMoreTokens())
                {
                    // write the output from the destem method
                    // to the output file
                    outputText.write(
newDestemObject.destem(st.nextToken(),hs));
                    outputText.newLine();
                } // end while(hasMoreTokens)
            } // end of try Lvl 2

            catch(IOException e)
            {
                System.out.println("IO Exception");
            }
        }
    }
}

```



```

String ampsOutFile =
    "sourceText/ampsOutFile.txt";
// output text file
String ampsoutputFile =
    "sourceText/ampsoutputFile.txt";
// output text file
String jmpsoutputFile =
    "sourceText/jmpsoutputFile.txt";
// input matching text file
String manualMatchFile =
    "sourceText/AMPS JMPS Macro Matching.txt";
//
    "sourceText/AMPS JMPS Matching.txt";
    "sourceText/spec Match File.txt";
// input matching text file
String outputManualMatchFile =
    "sourceText/AMPS JMPS 1st Pass.txt";
// input matching text file
String validWordFile =
    "sourceText/validwords.txt";
String jmpsPart1 =
    "sourceText/JMPS Part 1.txt";
String jmpsPart2 =
    "sourceText/JMPS Part 2.txt";
String jmpsPart3 =
    "sourceText/JMPS Part 3.txt";
String recallPrecisionOutput =
    "sourceText/recallPrecisionOutput.txt";
String jmpsPart4 =
    "sourceText/JMPS Part 4.txt";
String jmpsPart5 =
    "sourceText/JMPS Part 5.txt";
String jmpsPart6 =
    "sourceText/JMPS Part 6.txt";
String jmpsPart7 =
    "sourceText/JMPS Part 7.txt";
String jmpsPart8 =
    "sourceText/JMPS Part 8.txt";
String jmpsPart9 =
    "sourceText/JMPS Part 9.txt";
String outputMatchList =
    "sourceText/outputMatchList.txt";
String manualMatchOutputFile =
    "sourceText/manualMatchOutputFile.txt";
String manualMacroMatchOutputFile =
    "sourceText/manualMacroMatchOutputFile.txt";
String outputMacroManualMatchFile =
    "sourceText/outputMacroManualMatchFile.txt";
String matchOutputFile =
    "sourceText/matchOutputFile.txt";
Hashtable jmpsWordTable = new Hashtable();
Hashtable ampsWordTable = new Hashtable();
Hashtable unalteredReqList = new Hashtable();

/*
    // create a stop word list in a hash table
    boolean stopFlag = true;
    fileReadAndParse( pPStopList, jmpsWordTable, stopFlag);

    String JMPSReqDelimString = "JMPS-0";
    char[] reqNumPunct = {'.', '-'};
    String adjReqNum = "";
    buildUnalteredReqList(sourceTestFile, unalteredReqList,
        adjReqNum, reqNumPunct, JMPSReqDelimString);

    // tokenize the source document
    String delimString = "~!@#$%^&*()+=|{}[]:;<,>.? \\t\\n\\r\\f\\\"";
    //set this value to one "1" to change all tokens to lower case
    // except acronyms.
    int lowerCaseSelect = 1;
    preProcess(sourceTestFile, newTestFile, delimString,
        jmpsWordTable,
lowerCaseSelect);

```

```

destemPreProcess(newTestFile, validWordFile, jmpsTestFile);

        // add tokens from the source file to the hashtable
        stopFlag = false;
fileReadAndParse(jmpsTestFile, jmpsWordTable, stopFlag);

// add unique idenitifers to the requirement number string
String searchString = "3.";
String appendString = "amps-";
String appendString = "AMPS-";
appendPreProcess(ampsTestFile, intAmpsTestFile, searchString,
        appendString);

// create another stop word list hash table
stopFlag = true;
fileReadAndParse( ppStopList, ampsWordTable, stopFlag);

// tokenize the next source file
delimString = "`~!@$%^*()+=|{}[];';<, >. ? \t\n\r\f\"";
preProcess(intAmpsTestFile, ampsOutFile,
        delimString, ampsWordTable, lowerCaseSelect);

destemPreProcess(ampsOutFile, validWordFile, finalAmpsTestFile);

// add the tokens to the hash table
stopFlag = false;
fileReadAndParse(finalAmpsTestFile, ampsWordTable, stopFlag);

        // add unique idenitifers to the requirement number string
String newSearchString = "3.";
String newAppendString = "AMPS-";
appendPreProcess(manualMatchFile, outputManualMatchFile,
        newSearchString, newAppendString);
*/
// create the JMPS DocObject shell
DocObject jmpsDocument = new DocObject(jmpsTestFile);

// build a docObject using the two strings to indicate the token
// delimiters and the identifying string for the start of a requirement
String newdelimString = "\n";
String reqDelimString = "JMPS-0";
MasterMacroRequirement JMPSMasterMacro = new
        MasterMacroRequirement(reqDelimString);

reqPreProcess(jmpsTestFile, newdelimString, reqDelimString,
        jmpsDocument, JMPSMasterMacro, true);

JMPSMasterMacro.updateMacroPVals(jmpsDocument);

System.out.println("JMPS Complete");

// create the AMPS DocObject shell
DocObject ampsDocument = new DocObject(finalAmpsTestFile);

// build a docObject using the two strings to indicate the token
// delimiters and the identifying string for the start of a requirement
newdelimString = "\n";
reqDelimString = "AMPS-3";
MasterMacroRequirement AMPSMasterMacro = new
        MasterMacroRequirement(reqDelimString);

reqPreProcess(finalAmpsTestFile, newdelimString, reqDelimString,
        ampsDocument, AMPSMasterMacro, false);

AMPSMasterMacro.updateMacroPVals(ampsDocument);

System.out.println("AMPS Complete");

System.out.println(" Begin Macro-Matching");

```

```

MacroMatchObject newMacroMatch = new MacroMatchObject(
    AMPSMasterMacro, ampsDocument, JMPSMasterMacro,
    jmpsDocument);

newMacroMatch.matchListOutput(matchOutputFile);

/*
String ampsDelimString = "AMPS-3";
String jmpsDelimString = "JMPS-0";
MacroMatchObject manualMacroMatchObject = new MacroMatchObject(
    AMPSMasterMacro, JMPSMasterMacro,
    outputManualMatchFile, ampsDelimString,
    jmpsDelimString);

manualMacroMatchObject.matchListOutput(manualMacroMatchOutputFile);

macroResultComparisonOutput(manualMacroMatchObject, newMacroMatch,
    recallPrecisionOutput, outputMatchList);
*/

System.out.println("Macro-Matching Complete");

System.exit( 0 );

    }
} // end of WordParseMacroReq

```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

- AMPS97 Aviation Mission Planning System (AMPS): System Sub-System Specification, 05 August 1997
- AORD97 AMPS Operational Requirements Document, 5 June 1997.
- ARAN93 Arango, G., Schon, E., Penttengill, R., "Design as evolution and reuse", *Proceedings Advances in Software Reuse., Selected Papers from the Second International Workshop on Software Reusability*, 1993, Page(s): 9 -18
- BATO98 Batory, D., "Product Line Architectures", *Invited Presentation: Smalltalk and Java in Industry and Practical Training*, Erfurt, Germany, October 1998.
- BERZ89 Berzins, V., Kopas, R., "A Student's Guide to Spec", Naval Postgraduate School, Monterey, CA.
- BERZ91 Berzins, V., Luqi, "Software Engineering with Abstractions," Addison-Wesley, 1991.
- BJOR98 Bjorner, D., "Domains as a Prerequisite for Requirements and Software Domain Perspectives & Facets, Requirements Aspects and Software Views", *International Workshop RTSE*, 1998, Page(s): 1 -42
- CLAR96 Clarke, E., Wing, J., ET AL., "Formal Methods: State of the rt and Future Directions", *ACM Computing Surveys Vol. 28, No. 4, December 1996*. Page(s): 626 -643
- COHE92 Cohen, S., Stanley, J., Peterson, A., Krut, R., "Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain", *Technical Report, CMU/SEI-91-TR-28 ESD-91-TR-28*, June, 1992.
- DAVI94 Davis, M., Hawley, H., "Reuse of software process and product through knowledge-based adaptation", *Proceedings., Third International Conference on Software Reuse: Advances in Software Reusability*, 1994 , Page(s): 44 -52
- DIAZ87 Prieto-Diaz, R., "Domain analysis for Reusability," *Proceedings of COMPSAC 87, The 11th Annual International Computer Software and Applications Conference*, cat.no. 87CH2447-1 OCT. 7-9, 1987, Page(s): 23 -29
- FRAK97 Frakes, W., Prieto-Diaz, R., Fox, C., "DARE-COTS. A domain analysis support tool", *Proceedings., XVII International Conference of the Chilean Computer Science Society*, 1997, Page(s): 73 -77
- FRAN95 France, R., Horton, T., "Applying Domain Analysis and Modeling: An industrial Experience", *Proceedings of the 17th International Conference*

- on Software Engineering, Symposium on Software Reusability, April 1995, Page(s) 206-214.
- FRAN97 Frankel, M., and Winant, B., "A Taxonomy for Domain Partitioning and Reuse", *Object Magazine*, March 1997.
- GRAH98 Graham, I., *Requirements Engineering and Rapid Development*, Addison Wesley Longman Limited, 1998
- GREE94 Greenspan, S., Mylopoulos, J., Borgida, A., "On Formal Requirements Modeling Languages:RML Revisited", *Proceedings. ICSE-16., 16th International Conference on Software Engineering*, 1994. Page(s): 135 -147
- HALL96 Hall, A., "Using formal methods to develop an ATC information system", *IEEE Software*, Volume: 13 2 , March 1996 , Page(s): 66 -76
- HAMI93 Hamilton, J., "SEE integration to support megaprogramming", *Proceedings., Software Engineering Environments Conference*, 1993, Page(s): 17 -22
- IBRA96 Ibrahim, O., *A Model and Decision Support Mechanism for Software Requirements Engineering*, Ph.D Dissertation, Naval Postgraduate School, September 1996
- INFO98 Information Assembly Automation Web Site, <http://www.infoauto.com/articles/sgml/markup-verify.htm>, 1998
- JACK95 Jackson, M, *Software Requirements and Specifications*, ACM Press, 1995
- JMPS99 Joint Mission Planning System(JMPS): System Sub-System Specification, 15 November 1999.
- JMPS99a Joint Mission Planning System Web Site, <https://jmps.chinalake.navy.mil>, November 1999.
- KOTO98 Kotonya, G., Sommerville, I., *Requirements Engineering: Process and Technique*, John Wiley & Sons Ltd., 1998
- LARM97 Larman, C., *Applying UML and patterns: an introduction to object-oriented analysis and design*, Prentice-Hall, Inc., 1997
- LIM98 Lim, W., *Managing Software Reuse*, Prentice Hall, Inc., 1998
- LUQI97 Luqi, and Goguen, J., "Formal Methods: Promises and Problems", *IEEE Software*, Vol. 14 No. 1, Jan 1997, Page(s): 73-85.
- MAID91a Maiden, N., and Sutcliffe, A., "Analogical Matching For Specification", *Proceedings., 6th Annual Knowledge-Based Software Engineering Conference*, 1991, Page(s): 108 -116
- MAID91b Maiden, N., and Sutcliffe, A., "Reuse of analogous specifications during requirements analysis", *., Proceedings of the Sixth International Workshop on*

- Software Specification and Design*, 1991, Page(s): 220 -223
- MAID93a Maiden, N., and Sutcliffe, A., "Case-based reasoning in software engineering", *IEE Colloquium on Case-Based Reasoning*, 1993 , Page(s): 2/1 -2/3
- MAID93b Maiden, N., and Sutcliffe, A., "A computational mechanism for parallel problem decomposition during requirements engineering", *Proceedings of the 8th International Workshop on Software Specification and Design*, 1996 , Page(s): 159 -163
- MAID93c Maiden, N., and Sutcliffe, A., "People-oriented software reuse: the very thought", *Proceedings Advances in Software Reuse., Selected Papers from the Second International Workshop on Software Reusability*, 1993, Page(s): 176 -185
- MAID94a Maiden, N., and Sutcliffe, A., "Requirements Critiquing Using Domain Abstractions", *Proceedings of the First International Conference on Requirements Engineering*, 1994, Page(s): 184 -193
- MATH00 Mathews, Hunter, Software Engineer, Interview by author, May 31, 2000, Huntsville, AL.
- MILS96 MIL-STD-2525A, "Department of Defense Interface Standard: Common Warfighting Symbology", 15 December 1996.
- NEIL98 Neil, M.; Ostrolenk, G.; Tobin, M.; Southworth, M., "Lessons from using Z to specify a software tool", " *IEEE Transactions on Software Engineering*", Volume: 24 Issue: 1 , Jan. 1998 Page(s): 15 -23
- PORT80 Porter, M.F., "An Algorithm for Suffix Stripping", *Program*, Vol.14, 1980, Page(s): 130-137
- ROWE99 Rowe, N., "Precise and Efficient Retrieval of Captioned Images: The MARIE Project", *Library Trends*, Vol. 45, No. 2, Fall 1999, Page(s): 475-495
- SALT88 Salton, G., Buckley, C., "Term-Weighting Approaches in Automatic Text Retrieval", *Information Processing and Management*, Vol. 24, 1988, Page(s): 513-523
- SMIT92 Smith, T., "READS: a requirements engineering tool ", *Proceedings of IEEE International Symposium on Requirements Engineering*, 1993, Page(s): 94 -97
- SOMM97 Sommerville, I., and Sawyer, P., *Requirements Engineering, a good practice guide*, John Wiley & Sons, Inc., 1997.
- SUN99 Sun Microsystems Web Site, Java 2 Platform Standard Edition vl.3,
<http://java.sun.com/j2se/1.3/docs/api/index.html>, April 2000.

- SUTC94 Sutcliffe, A., and Maiden, N., "Domain modeling for reuse", *Proceedings., Third International Conference on Software Reuse: Advances in Software Reusability* , 1994 , Page(s): 169 -177
- SUTC98 Sutcliffe, A., Maiden, N., "The domain theory for requirements engineering", *IEEE Transactions on Software Engineering*, Volume: 24 3 , March 1998 , Page(s): 174 -196
- W3C98 W3C Recommendation, Extensible Mark-up Language(XML) 1.0, <http://www.w3.org/TR/REC-xml> February 1998.
- WALE99 Wales, Terry, Senior Engineer, System Dynamics Incorporated, PM AEC, Huntsville AL. Interview by author, Huntsville, AL., 13 Oct 99.

INITIAL DISTRIBUTION LIST

- | | | |
|----|---|---|
| 1. | Defense Technical Information Center
8725 John J. Kingman Rd., STE 0944
Fort Belvoir, Virginia 22060-6218 | 2 |
| 2. | Dudley Knox Library, Code 52
Naval Postgraduate School
411 Dyer Rd.
Monterey, California 93943-5101 | 2 |
| 3. | Dr. Dan Boger, Code CS
Chariman, Department of Computer Science
Department of Computer Science
Monterey, California 93943-5118 | 1 |
| 4. | Dr. Man-Tak Shing, Code CS/Sh
Naval Postgraduate School
Department of Computer Science
Monterey, California 93943-5118 | 2 |
| 5. | Dr. Neil Rowe, Code CS/Rp
Naval Postgraduate School
Department of Computer Science
Monterey, California 93943-5118 | 1 |
| 6. | Mr. Terry Wales
PM, AEC
950 Explorer Blvd.
Huntsville, AL 35806 | 1 |
| 7. | Captain Eric Stierna
SCI-TECH, Box 131
Unit 45015
APO, AP 96338-5015 | 1 |